

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**A PATTERN-MATCHING APPROACH FOR AUTOMATED
SCENARIO-DRIVEN TESTING OF STRUCTURED
COMPUTATIONAL POLICY**

by

Mehmet Sezgin

September 2001

Principal Advisor:
Associate Advisor:

James Bret Michael
Richard Riehle

Approved for public release; distribution is unlimited

Report Documentation Page

Report Date 30 Sep 2001	Report Type N/A	Dates Covered (from... to) -
Title and Subtitle Generation of Test Plans for a Policy Workbench		Contract Number
		Grant Number
		Program Element Number
Author(s) Sezgin, Mehmet		Project Number
		Task Number
		Work Unit Number
Performing Organization Name(s) and Address(es) Research Office Naval Postgraduate School Monterey, Ca 93943-5138		Performing Organization Report Number
Sponsoring/Monitoring Agency Name(s) and Address(es)		Sponsor/Monitor's Acronym(s)
		Sponsor/Monitor's Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited		
Supplementary Notes		
Abstract		
Subject Terms		
Report Classification unclassified		Classification of this page unclassified
Classification of Abstract unclassified		Limitation of Abstract UU
Number of Pages 158		

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2001	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: Automatic Generation of Test Plans for a Policy Workbench			5. FUNDING NUMBERS	
6. AUTHOR(S) Sezgin, Mehmet				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>Organizations are policy-driven entities. Policy bases can be very large and complex; these factors are in the dynamic nature of policy evolution. The mechanical aspects of policy modification and assurance of the consistency, completeness, and correctness of a policy base can be automated to some degree. Such support is known as computer support for policy.</p> <p>We developed an object-oriented schema-based approach to structure policy. Our structural model consists of Unified Modeling Language class and collaboration diagrams. The structural model is used by a suite of testing tools. We present a case study to illustrate our approach to automated testing of policy.</p> <p>Our approach to test-case generation is based on the use of patterns within policy statements and relationships between policy objects. The test spectrum has query-specific tests at one end, and the generic types of tests at the other end. We introduce the use of statistical inference to reuse test cases by determining the patterns that approximate the query-to-be-executed. Query mapping, anytime reasoning and fuzzy logic concepts in policies and their applications are discussed.</p>				
14. SUBJECT TERMS Policy, policy workbench, testing, policy testing, test strategy, test case, test design, UML, deontic logic, OTTER, anytime reasoning, test patterns, query mapping, software, work-flow, query, proof, collaboration, logic, security, formal methods, computer, security.			15. NUMBER OF PAGES 144	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**A PATTERN-MATCHING APPROACH FOR AUTOMATED SCENARIO-
DRIVEN TESTING OF STRUCTURED COMPUTATIONAL POLICIES**

Mehmet Sezgin
First Lieutenant, Turkish Army
B.S., Turkish Military Academy, 1996

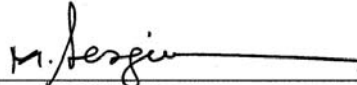
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

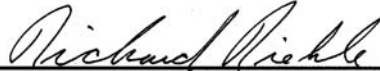
**NAVAL POSTGRADUATE SCHOOL
September 2001**

Author:


Mehmet Sezgin

Approved by:


James Bret Michael, Principal Advisor



Richard Riehle, Associate Advisor



Chris Eagle, Chairman
Computer Science Department

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Organizations are policy-driven entities. Policy bases can be very large and complex; these factors are in the dynamic nature of policy evolution. The mechanical aspects of policy modification and assurance of the consistency, completeness, and correctness of a policy base can be automated to some degree. Such support is known as computer support for policy.

We developed an object-oriented schema-based approach to structure policy. Our structural model consists of Unified Modeling Language class and collaboration diagrams. The structural model is used by a suite of testing tools. We present a case study to illustrate our approach to automated testing of policy.

Our approach to test-case generation is based on the use of patterns within policy statements and relationships between policy objects. The test spectrum has query-specific tests at one end, and the generic types of tests at the other end. We introduce the use of statistical inference to reuse test cases by determining the patterns that approximate the query-to-be-executed. Query mapping, anytime reasoning and fuzzy logic concepts in policies and their applications are discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	BACKGROUND	1
B.	MOTIVATION	2
C.	HYPOTHESIS.....	2
D.	SCOPE	2
E.	SUMMARY OF APPROACH.....	3
II.	UML IN THE DESCRIPTION AND STRUCTURE OF THE ARCHITECTURE OF POLICY.....	7
A.	UML SUPPORT FOR MODELING POLICY	7
1.	The Advantages of Using The Formal Specifications in UML	9
2.	The Expectations from UML Formalism	9
a.	<i>Executability.....</i>	<i>9</i>
b.	<i>Understandability.....</i>	<i>10</i>
c.	<i>Extensibility.....</i>	<i>10</i>
d.	<i>Modifiability</i>	<i>10</i>
3.	Why do We Need Formalism in UML?	12
B.	POLICY REPRESENTATION.....	12
1.	What is Included in the Policy?	12
2.	Policy Representation in UML	14
a.	<i>The Difficulties of Representing Policies in UML</i>	<i>15</i>
3.	Ways of Representing Policy Using UML as a Framework.....	15
a.	<i>Defining Policies as Classes</i>	<i>15</i>
b.	<i>Defining Policies as Communities</i>	<i>17</i>
c.	<i>Defining Polices as Diagrams</i>	<i>17</i>
4.	Deontic Logic as a Policy Structuring Method.....	17
III.	CASE STUDY	21
A.	CASE STUDY DESIGN.....	21
1.	Rule 1.....	22
2.	Rule 2.....	22
3.	Rule 3.....	22
4.	Rule 4.....	23
5.	Rule 5.....	23
6.	Rule 6.....	25
7.	Rule 7.....	25
8.	Rule 8.....	26
9.	Rule 9.....	26
10.	Rule 10.....	27
11.	Rule 11.....	27
12.	Rule 12.....	28
B.	REPRESENTING POLICY VIA UML DIAGRAMS.....	28

C.	USING COLLABORATION DIAGRAMS FOR UML REPRESENTATION	30
1.	Specification-level Collaboration Diagrams	30
2.	Instance-level Collaboration Diagrams.....	30
D.	WHAT ARE THE TEST CASES GIVEN THESE RULES?	32
1.	Rule 1.....	32
2.	Rule 2.....	33
3.	Rule 3.....	34
4.	Rule 4.....	34
5.	Rule 5.....	34
6.	Rule 6.....	35
7.	Rule 7.....	35
8.	Rule 8.....	36
9.	Rule 9.....	36
10.	Rule 10.....	36
11.	Rule 11.....	36
12.	Rule12.....	37
E.	HOW CAN WE HANDLE DIFFERENT QUERIES AT THE TEST SPECTRUM?	39
F.	QUERY MAPPING	42
G.	FUZZY LOGIC AND POLICY TESTING.....	46
1.	What does Fuzzy Logic Offer?	46
2.	What can be the Fuzzy Concepts in the Policy Workbench?	47
3.	What Techniques of Fuzzy Logic can be Applied to Policy Testing?	47
H.	ANYTIME REASONING IN POLICIES	48
I.	TARGET TEST PATTERN SELECTION	50
J.	EXAMPLE OF TARGET PATTERN SELECTION.....	52
K.	EXECUTION OF THE IMPLEMENTATION	53
1.	Query 1.....	55
2.	Query 2.....	56
3.	Query 3.....	57
4.	Query 4.....	58
5.	Query 5.....	59
6.	Query 6.....	61
IV.	POLICY TESTING	65
A.	INTRODUCTION TO POLICY	65
B.	POLICY AS A MOVING TARGET.....	65
C.	MAINTENANCE OF POLICY.....	66
D.	TESTING.....	67
1.	What are the Steps When Testing Policy?.....	68
2.	Development of test cases in policy testing	68
E.	TEST PATTERNS AND REUSABILITY	70
F.	STRATEGY FOR DEVELOPING TEST CASES	71
G.	PROPOSED APPROACH TO POLICY TESTING.....	74

1.	Unautomated Testing Process.....	75
2.	Automated Policy Testing Process (Our Testing Strategy).....	76
3.	Benefits of Our Testing Approach.....	80
V.	RELATED WORK.....	81
A.	DISCUSSION.....	81
1.	Policy Specification and Axiomization.....	81
2.	Policy Testing.....	82
3.	UML as Policy Representation Framework.....	84
4.	Formal Methods in Policy Structuring and Testing.....	87
VI.	CONCLUSIONS AND FUTURE WORK.....	91
A.	CONCLUSIONS.....	91
B.	FUTURE WORK.....	93
APPENDIX A	OTTER TUTORIAL.....	95
A.	HIGHLIGHTS OF OTTER.....	95
B.	SET OF SUPPORT STRATEGY (SOS).....	97
C.	HOW DOES THE SET OF SUPPORT STRATEGY WORK?.....	99
APPENDIX B	WORKFLOW MODELING WITH NORM ANALYSIS.....	101
APPENDIX C	SOURCE CODE OF THE IMPLEMENTATION.....	103
APPENDIX D	QUERIES AND PROOFS.....	115
APPENDIX E	IMPLEMENTATION DIAGRAMS.....	125
APPENDIX F	GLOSSARY.....	133
	LIST OF REFERENCES.....	135
	INITIAL DISTRIBUTION LIST.....	139

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1.	The Need for UML Diagram Tranformations.	8
Figure 2.	The Formalism Goals Expected from UML.	11
Figure 3.	Policy Class in Policy Framework.	16
Figure 4.	UML Diagram of the Overall Implementation Schema.	29
Figure 5.	Test Pattern Distribution Diagram.	41
Figure 6.	Query to Rules Mapping Diagram.....	44
Figure 7.	Inter-query Relationship Diagram	44
Figure 8.	Intra- relationship Diagram.....	45
Figure 9.	Semantic Mapping Diagram	45
Figure 10.	Target Test Pattern Region Selection Diagram	51
Figure 11.	Sampling Diagram Within Selected Target Regions Diagram.....	51
Figure 12.	Resolution-style Theorem Prover Workflow Diagram.....	54
Figure 13.	Test Case Specification Diagram.....	69
Figure 14.	Engineering Diagram of Our Automated Testing Process.	79

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Policy Test Property Distribution Table	38
Table 2.	Representation of Symbols and Operators in OTTER.....	97
Table 3.	Representation of Formulas in OTTER.....	97

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First of all, I send my gracious thanks to my advisors, Professors Bret Michael and Richard Riehle for their patience, knowledge, and guidance during my research. Both of them turned my education into a unique and joyful experience.

Drs. Larry Wos and William McCune, of Argonne National Laboratory and Professor Michael Beeson from Santa Jose State University provided me with guidance in working with automated reasoning systems. Dr. Frederic Cuppens of ONERA-CERT, shed light on the process of structuring policy and provided me with a set of policy for testing my hypothesis.

Lastly, I thank the Turkish Armed Forces for the opportunity to pursue a master's degree at the Naval Postgraduate School so that I can better serve my country.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

Organizations are policy-driven entities [36]. An organization's policy defines when and in what context the policy is applicable, what information must be available for the policy to be used and enforced, what to do if a policy is violated or is inconsistent with another policy, and how to evaluate adherence to policy [8].

Policies are statements of goals, or rules or guidance governing the actions taken to satisfy goals[36]. They provide broad direction and goals. The “policy” term is synonymous with preferred behavior.

Policies may be implicit (i.e., unwritten) or explicit, and govern the behavior of the actors for whom the policy applies. There is a natural hierarchy of policy. At the highest level is meta policy: policy about policy. For example, a meta policy might specify when lower level policies should be enforced. Policy can be decomposed into successive finer levels of abstraction, until at some point the refinement yields one or more system requirements.

Policy bases can be very large and the relationships between policies can be complex [36]. An ad hoc approach to structuring policy might add more complexity to the existing policy base. Moreover, checking for gaps in policy or analyzing the consequences of changing policy in an unstructured policy base turns policy testing into a computationally hard problem. The main reason behind this problem is the difficulty of creating a policy test suite that can check for inconsistencies, incompleteness, and other gaps.

Merging, reorganization, restructuring and downsizing efforts in an organization or among organizations can change one or more of its policies. Policies must be structured in such a way that enables seamless integration after merging, reorganization or downsizing.

A policy workbench—an integrated set of computer-based tools for developing, reasoning about, and maintaining policy—is intended to assist the makers and users of policy.

B. MOTIVATION

The motivation for our research stems from the need to automate the testing of policy. The tasks involved in using the diverse array of tools within a policy workbench for testing could be viewed by a user as difficult or even insurmountable. Testing policy in its computation form requires a certain level of technical expertise (e.g., a knowledge of formal methods) and involves information-processing tasks that are best suited to be mechanized rather than performed manually. It is impossible to prove the absence of gaps in policy, so one must satisfy by demonstrating that the policy is free of gaps for specific test cases. The challenge is to determine which test cases to run and what the composition of those test cases should be. In this thesis, we introduce a systematic and repeatable process for automatically testing policy.

C. HYPOTHESIS

Our hypothesis is that it is possible to automatically generate test cases for operational policy by utilizing test patterns.

D. SCOPE

The primary focus of this thesis is on automatically testing for logical consistency among operational policies. We limit our investigation of test-case generation to support for using a first-order resolution-style theorem prover. In addition, we only consider the use of object-oriented structural models of policy statements from which to search for test patterns.

E. SUMMARY OF APPROACH

The approach employed in this thesis consists of the following:

- Conduct a thorough literature survey and analyze different approaches on structuring and developing policy, and testing the policy workbench.
- Create an object-oriented schema of the policy base, using the Unified Modeling language to document the model.
- Develop test cases, and scenarios to perform regression testing to ascertain the consequences of modifying policy.
- Maintain relationships and linkages among policies using test patterns.
- Query the existing policy base using a pattern-driven method of testing.

Organizations are policy-driven entities. Before the policy is refined into requirements for the system, the policy base needs to be checked for gaps, such as inconsistency or incompleteness. A policy workbench can be used to represent, reason about, maintain, implement, and enforce policy[36]. A policy workbench is an integrated suite of tools. [53] is one of the examples that uses automation to maintain, reason about, refine or enforce policy. Sibley, Michael, and Wexelblat propose a five-class architecture for a generic policy-workbench [52]. The policy base used in this thesis consists of operational policies. Operational policy can be represented as if-then (i.e., conditional) rules. For instance, if the actor is an agent,

and the head of the sending office, then she is obliged to sign the consignment note attached to the secret document.

In this thesis, we present a systematic way of structuring and developing policy using the Unified Modeling Language (UML); UML is a notation for expressing object-oriented designs. The policy maker enters policy, the policy workbench maintains a current data dictionary, tests the consistency of policy statement in relation to extant policy, and proposes scenarios for feedback. Michael, Ong and Rowe developed a natural-language input-processing tool (NLIPT) that translates policy into an object-oriented schema and formal models [53], relieving the user of the policy workbench from the error-prone task of manually generating the formal models.

Secondly, regression testing is performed against criteria such as correctness and consistency(i.e.,second user class). Test cases are developed based on the following test patterns: temporal, sequence, and counting aspects of the policy. The goal of the testing in this phase is to ascertain the consequences of modifying the policy base. These changes include adding a new policy, deleting an existing policy, composing policy, and modifying an existing policy.

Lastly, the policy base is analyzed via queries. A repository of queries is established. Each user query is associated with a test or a suite of tests. The policy user can reach this repository of queries through an interface. Maintaining queries in a repository also helps the policy user structure future queries based on previous ones and reuse the queries and query results. For instance for update operation, the update operation is converted into a query(or set of queries), then

evaluated using the testing techniques described in this thesis. Moreover, an approximation to dead-end queries can be obtained from the repository which gives the user an idea about the result of the query.

THIS PAGE INTENTIONALLY LEFT BLANK

II. UML IN THE DESCRIPTION AND STRUCTURE OF THE ARCHITECTURE OF POLICY

A. UML SUPPORT FOR MODELING POLICY

Modeling is a crucial part of all of the activities that lead up to the development of good software. Most object-oriented methods provide a family of graphical notations for creating abstract models of a system's behavior. The Unified Modeling Language (UML) [17] has become a widely used notation for expressing object-oriented designs. UML is a language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems [31]. The common interest is to use UML to create common interchangeable models for facilitating distributed design. UML uses a viewpoint-based description [8] with a family of diagrams, each of which is tailored for representing some aspect of the system under test(SUT).

Most of the work reported on UML defines and explains UML in terms of the construction of complete specifications. Despite the richness of constructs for modeling constraints, the language does not provide direct support for representing policy. In other words, the language's support for representing behavior does not provide for the creation of parameterized specifications or creation of related specifications by substituting specific policies into a framework. Linington [8] asserts that the expressive power can be provided by using UML as a base.

UML provides different types of diagrams types supporting the development process, from requirements specification to implementation. The UML diagrams support the creation of different views of the same system and modeling the system at multiple levels of abstraction. By using the four-tier architecture of the language [12] (i.e., user objects, model, metamodel, and metamodel) policy objects at different abstraction levels can be specified. Moreover, the language enables the transformation of UML diagrams into other UML diagrams. The beauty of this transformation is the ease of transformation at different levels of the policy-object hierarchy.

Policy tends to be dynamic in nature. As the organization's policy changes, the updates to the policy need to be modeled in order for the user of the policy workbench to reason about the changes. The representation of policy using UML notation results in common models of policy, which can then be refined into lower level system artifacts, thus providing a common representation for tracing between levels of refinement and the effects of changes throughout the policy base. The transformation of diagrams can be used in many ways [12], as shown in Figure 1.

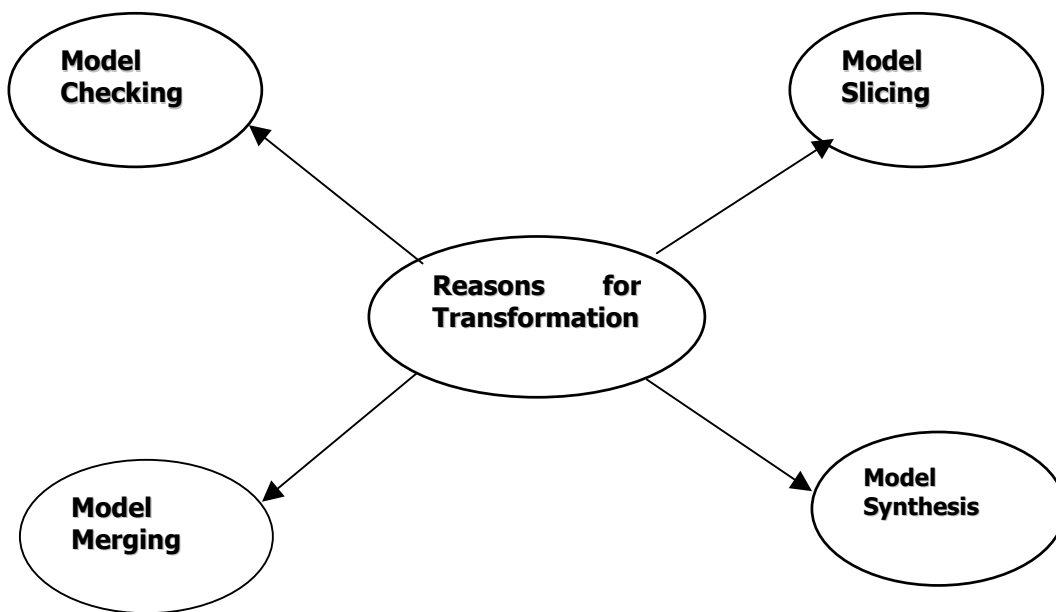


Figure 1. The Need for UML Diagram Transformations.

The policy objects represented at different abstraction levels using UML can be incorporated into different policy objects at different abstraction levels via the extensibility feature of the language.

1. The Advantages of Using The Formal Specifications in UML

The lack of formal semantics for object-oriented modeling notations can limit their use in the development of large, complex, or critical systems. Formal software specifications in UML have the following advantages [11]:

- Formal specification describes the behavior and structure being modeled free from most of the implementation details
- Formalizing an object-oriented model can reveal gaps, ambiguities, and inconsistencies
- Identification and retrieval of components can be partially automated
- Formal specifications and their associated formal system provide a basis for automated forward engineering (i.e., the creation of code from a model) and reverse engineering (the process of transforming code into a model through a mapping from a specific implementation language)

In addition to the advantages stated above, formal specifications built into the UML diagrams ease the process of verification and validation of models. In other words, the definition of the formal semantics of UML adds precision to the model and supports the use of advanced tools.

2. The Expectations from UML Formalism

The choice of the formalism depends on what the user expects from the formal semantics. The support expected from UML can be stated as follows [18], as shown in Figure 2:

a. Executability

This goal emphasizes the need for an operational formal semantics profile for UML. To give complete semantics to UML means to say exactly for every particular UML model, how the UML model is expected to behave at run-time and what are the semantics of the elements of the UML model. One way to

provide a complete operational semantics is to adopt the use of Abstract State Machines (ASM), which are composed of both static and dynamic semantics [18]. All of the models share a common part that contains the description of the basic principles (i.e., class, operation, attribute). Thus, a part of the static semantics can be realized by expressing in ASM the UML meta-model and well-formedness rules defined in the UML standard. The dynamic semantics are based on a set of behavior primitives such as time and communication. Finally, by using static and dynamic semantics, a symbolic execution of an UML model can be performed.

b. Understandability

Understandability is required to use UML correctly. Formalism with artifacts in UML is combined to precisely express policy and other system artifacts. Formalization for the sake of formalism does not necessarily enhance a model's understandability. The beauty of object-oriented design methodologies is their visual, intuitively appealing, modeling notations [17]. Thus, formalization efforts should increase but not degrade these powerful characteristics. For instance, Shroff and France have built formalism into UML, but the complexity of their resulting models makes these models difficult to understand. On the other hand, loose interpretation of the diagrams can convey different meanings to different users. Errors in interpretation (i.e., confusion and disagreement over the precise meaning of a model) can occur when hard-to-understand graphical notation.

c. Extensibility

It should be possible to add new concepts with a minimum effort. Policies are dynamic objects [15]. In our work, precisely specified policy objects with full semantics not only increase the understanding of the current policy objects but also eases the task of updating the model of policy.

d. Modifiability

The changes to policy should have a minimal impact on the rest of the semantics. The formalism must provide for ease of modifying and testing policy. As the policy objects are already defined in formal specifications, the

policy operations can be tested without actually implementing them. The results of testing enable to determine the inconsistencies(i.e., logical contradictions), violations and harmony(i.e., the seamless integration of multiple policy bases without any consistency). These and other gaps need to be resolved prior to baselining changes to policy.

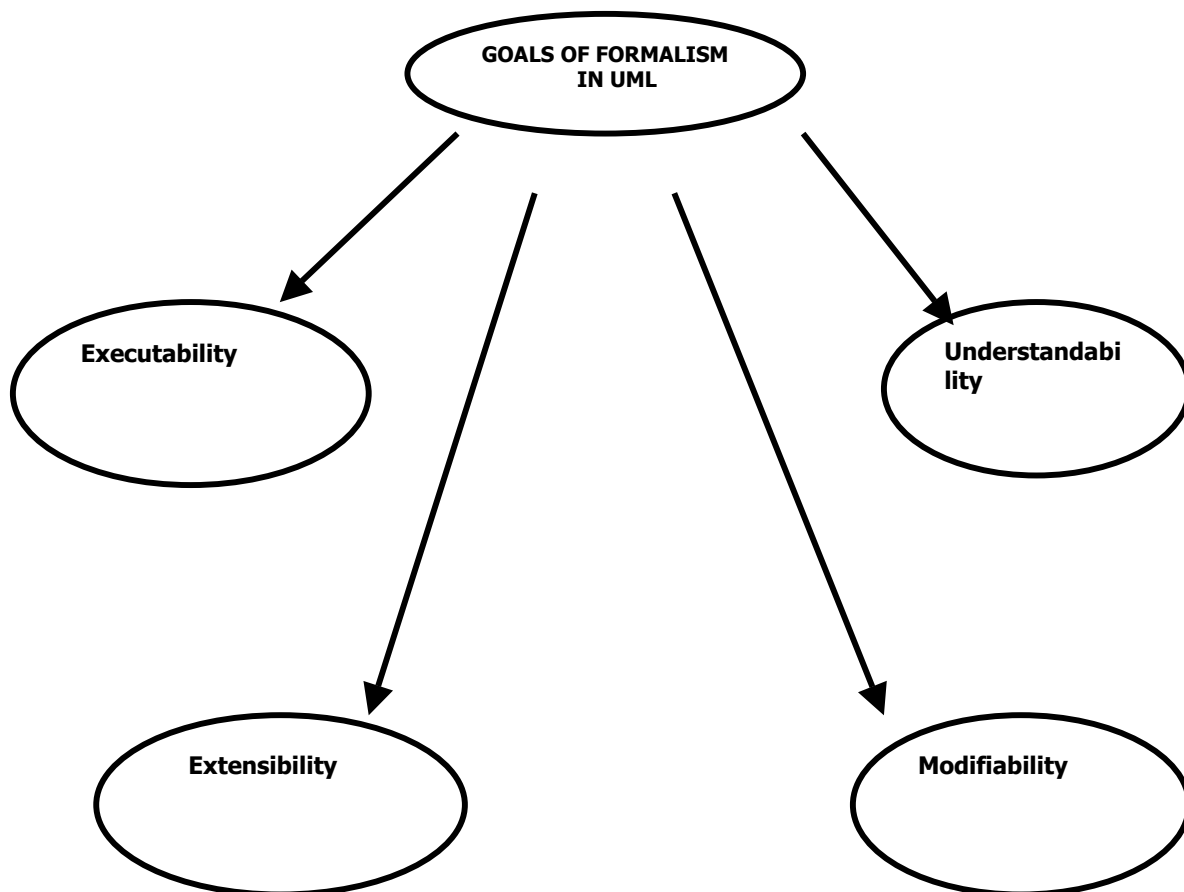


Figure 2. The Formalism Goals Expected from UML.

3. Why do We Need Formalism in UML?

An important part of UML is its semantics. Meta-models are used to describe the syntax of UML's static and behavioral models, while semantic details are expressed in informal English [10]. Unfortunately, the informal nature of these semantics is inadequate for justifying the use of formal analysis techniques with UML models.

B. POLICY REPRESENTATION

1. What is Included in the Policy?

A policy statement can define when the policy is applicable, what information must be available for the policy to be used and enforced, defining the decision process of what to do if a policy is violated or is inconsistent with another policy, how to evaluate adherence to the policy, and defining all invariants that must be adhered to for the policy to be effective [8]. In short, policy statements express constraints on the actions that constitute behavior within or between systems.

According to Linington, policy articulation involves the following:

- Defining a set of circumstances in which the policy is to apply
- Identifying some non-trivial choice to be made under the control of the policy
- Identifying what information that must be available for the policy to be interpreted
- Defining a decision procedure to be applied in assessing the situation and in actually making the choice

- Defining any invariants that may need to be respected by the system in general for the policy to be effective

Very few policies apply universally, without any preconditions. For instance, an access control policy must be clear as to what kind of access is being controlled, and whether the policy applies at all times, or only within some range of hours.

The need for invariants emerges from the idea to preserve the object during actions. For example, consider the policy “Classified documents can only be accessed by agents with adequate clearances.” This would be ineffective if anyone could define a local procedure for changing the classification of documents; so an invariant is needed which states that the parties other than the sender cannot change the classification level of the document. As an alternative to invariants, one could also apply the constraint on classification as a prohibition. The policy “Any agent who is not the sender of a classified document is prohibited to change the classification of this document” resolves this problem by directly forbidding the unwanted behavior by stating it as a separate rule instead of using invariants.

“What to include into the policy and what not to” [2] is the key component that eases or hardens the automation of representing policy. Cuppens in [2] stated that the complexity and ambiguity of the policy at hand is the most cumbersome and time-consuming problem to resolve during the specification of policy. To formalize the policy, their first task was to clarify the subtleties contained in the policy document written in an “administrative natural language.” The efforts to formalize a badly written policy is to get a correct and as precise as possible interpretation of the policy at hand. Moreover, the formalization of policy required the experts to know the administrative language used in the policy. The experts knowing the organizational administrative language can extract the definitions and normative rules from the policies without missing any rules.

As a solution to ambiguous policies, [11] and [1] proposed automated forward engineering or a schema-based approach (i.e., the formalization of policy

from a structural model of policy). Michael et al. found out that during the axiomatization process, a schema-based approach produced fewer errors in formalizing security policy than the non-schema-based approach. Using a model starting from the creation of policy in natural language will make clear what to include in the policy, decrease the formalization effort needed, and prevent structuring errors.

Who makes the policy and the styles for making the policy are out of the scope of this thesis; see [8] for more information about the actors and the styles.

2. Policy Representation in UML

As Linington states in [8], there are many possible ways policies can be formulated. In this thesis, we explore the main features of identifying a number of kinds of policy on different views.

The first step is to sort the things out that need to be specified in the policies. For clarity, conciseness and preciseness of the information in the specification, the following is a list given in [8]:

- Configure the constraints by controlling the cardinality of associations, or by constraining values reached by the navigation. Some configuration constraints cannot be expressed by cardinalities alone.
- Pieces of behavior can be specified by a use-case, or by a behavior, which is non-deterministic or has many possible transitions.
- Pre/post conditions can be added to actions, making the outcome of part of behavior more specific. In other words, this means prohibiting non-compliant actions.
- Internal structure can be added to states, providing detailed procedure for continuing actions within a high-level state.

- Optimization policies can be established for some or all of the behavior. Refinement of the optimization conditions needs to express the community objective effectively.

As a side note to the list, the invariants are advantageous to use as long as the invariants relate reasonably closely to the granularity of the model. The approaches in the preceding list are based on the creation of the policies via the design-tool chain. Linington's remarks does not fit well with specific set of policies (that is, the interpretation of policies for which most of the parts are generated by interactions at "run-time").

a. The Difficulties of Representing Policies in UML

As discussed in [8] many aspects of a policy statement can be represented within UML by adding constraints or additional finer scale structure to an existing pure-UML-diagrams-only design. However, the following two issues should be addressed:

- How to declare the extra body of information as a policy
- How to express limits on the policies which will actually be acceptable in a given situation

Resolution of these issues is needed to capture the aspects of the development process that are beyond the scope of UML.

3. Ways of Representing Policy Using UML as a Framework

By taking UML as a framework, the difficulties of policy representation mentioned above can be overcome. Linington in [8] is the forerunner in the literature who clearly defines strategies for representing policy using UML. According to his classification, policy can be represented in three ways:

a. Defining Policies as Classes

The policy maker defines a class that clarifies the detailed behavior to express the policy. Just naming the class is too weak. As the policy objects are

in a hierarchical manner, the new class must be put in its place in this hierarchy. To meet that goal, the author of the main specification would provide upper and lower classes; see Figure 3. In addition to that, the author applies whatever constraints would be necessary and assigns a policy class to the policy objects in the package to form an inheritance chain. The upper classes define the policy object's environment, and the lower class is used to localize policy reference.

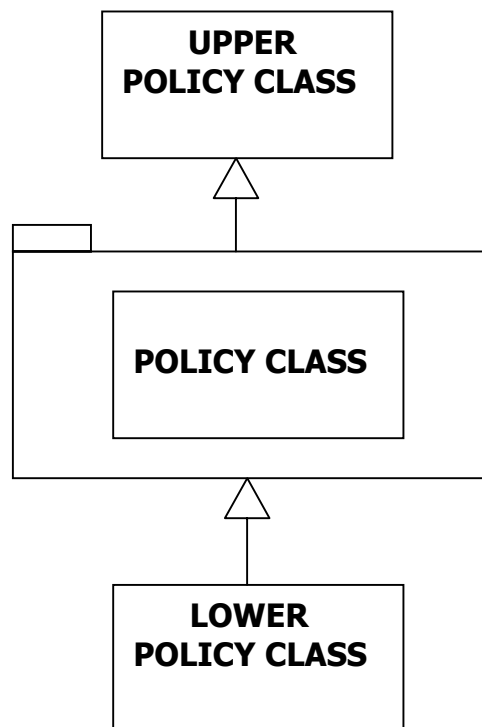


Figure 3. Policy Class in Policy Framework.

We represent policy objects as classes in this thesis. One of the reasons behind this decision is our goal to use UML as a tool to define and structure policy. Once the class diagrams of the policy objects are at hand, the detailed diagrams are derived from these class diagrams. This method permit us to

merge multiple diagrams. See an example of “defining policies as classes” in the Chapter III.

b. Defining Policies as Communities

A community is a set of interacting objects that have come together into a configuration so that they can interact to achieve some purpose. Communities are typed and their type determines the broad kinds of behavior they can exhibit, and the policies, which control their formulation, interaction and evolution [19]. Community is widely used in Open Distributed Processing (ODP). Communities can also be represented as collaboration diagrams; this way of representation is out of the scope of this thesis. For further information on this topic see [19].

c. Defining Policies as Diagrams

Defining policies as diagrams isolates each policy from the remainder of the specification by using the presentation structure. The challenge is to merge multiple diagrams into a single model to unify the material into a single specification. It is not clear that tool support for interpreting multiple diagrams is present. The advantage of this representation strategy is that it places very little restriction on the policy specification process. See Appendix E for examples of defining policy as diagrams of the case study in Chapter III.

4. Deontic Logic as a Policy Structuring Method

As the concepts of permission, prohibition and obligation are formally studied in deontic logic, it is suitable as a starting point for the specification of a normative system. A normative system is a system in which the behavior of the interaction among policy objects is governed by a set of norms [19]. The concepts of prohibitions, obligations and permissions can be referred to as deontic statements about a system. [19] used deontic logic for the Reference Model for Open Distributed Processing(RM-ODP) enterprise modeling and also [2] used it for specifying security policy. We also adopt deontic logic in this thesis for the following reasons:

- It represents a universal formal language, useful for expressing any kind of knowledge or data.
- It supports formal calculus methods.
- There is tool support (e.g., PROLOG, OTTER, and other automated reasoning programs).

Deontic logic has some limitations, not accounting for the interactions between agents. But such limitations do not affect the course of the case studies in this work.

Three deontic modalities are permission, obligation and prohibition. The RM-ODP defines the permission as “a prescription that a particular behavior is allowed to occur. A permission is equivalent to there being no obligation for the behavior not to occur.” Permission also can be defined by the following [20]:

- An action
- A participant-role in that action
- A predicate on social behavior
- A community-role
- An authority which grants the permission

The concept of permission can either be regarded as “having permission” or “granting permission.” The former relates to what actions may occur, while the latter involves an implicit or explicit agency, and there are consequential obligations on the authority as a result of granting permission. Thus, the authority should not normally simultaneously grant permission and prevent the permitted action from taking place. For instance, “If Agent_A is permitted to perform Action_B, then Agent_A has the right to choose whether to perform the action, but the authority has no such choice; it is obligated to allow Agent_A to perform Action_B.” The analysis of obligations implied by granting permissions is an area in which some level of conflict and inconsistencies in the obligation is inevitable.

These areas are fruitful for testing in the context of checking for logical inconsistencies between policies.

RM-ODP defines prohibition as “a prescription that particular behavior must not occur. A prohibition is equivalent to there being an obligation for the behavior not to occur”[20]. Prohibition is also defined similarly as follows:

- An action
- A participant-role in that action
- A predicate on social behavior
- A community-role
- An authority which grants the permission

If one of the agents has this prohibition, then the agent cannot play the participant-role in the action.

RM-ODP defines obligation as “a prescription that particular behavior is required. An obligation is fulfilled by the occurrence of the prescribed behavior [20].” Obligations are different from permissions and prohibitions. First of all, obligations are not granted or imposed, but rather are agreed as a part of joining a contract. Obligations are expressed as follows:

- Enable the trigger conditions such as a predicate that holds while the obligation is “active,” that is, if the rule is valid, then activate obligation or a pair of activating or deactivating conditions that toggle obligations into active and inactive modes (i.e., switch rule then case A or case B or case C or default).
- Satisfaction condition and violation condition also enables one to set obligations. For instance, standing obligations can never be satisfied: they must be defined by a violation condition.

The primary purpose of formal methods is to help engineers construct more reliable systems[26]. Graphical object-oriented (OO) models can be used to represent real-world concepts. UML has been forwarded as the common language for OO modeling. The beauty of using a common language is seamless transfer of models between design and analysis tools.

In summar, UML can be used for structuring the policy, policy objects and relationships between these objects.

III. CASE STUDY

A. CASE STUDY DESIGN

Our design of the case study was challenging for the following reasons: Real-world policy tends to be complex and ambiguous(i.e., left open to interpretation) when found in natural-language texts. Secondly, the administrative language used in a document has subtleties that to be taken into consideration when interpreting policy. In order to formalize the policy, the clarification of subtleties, ambiguities and imprecision is required. This may require consultation with a person who has significant amount of knowledge about the organization and policy formalization. The role of such a person is crucial. Thirdly, the natural language policy not only contains normative rules and definitions but also contains comments or advice, the latter ones of which are discarded in our case study for the sake of simplicity. To include policy that just consists of comments or advice is not straightforward to do. Deciding which one is a rule or which one is not is troublesome because a policy can take the form of advice. While trying to select the rules, the full semantics of the policy might get corrupted. Having only formalization knowledge is not enough to convert a natural language statement of policy into a formalized policy. [2] stated that creating a formal policy from a natural language policy is a time-consuming, cumbersome activity. Lastly, the intentional lack of precision in the specification of some policy in large organizations for the sake of application flexibility is also an impediment to formalization of policy.

In our case study, some policies can be viewed as conditional rules with normative conclusions. In other words, they can be translated by: if this condition is satisfied, then this agent is obliged, permitted, or prohibited to do something. As the automatic translation of natural-language statements of policy to a computational form is outside of the scope of this thesis, the case study developed is adapted from [2]. The authors in [2] used their own specification language. We chose to use first-order predicate logic with equality as the policy specification language. We also choose to use OTTER, a first-order resolution-style theorem

prover, as the testing tool. For further details on OTTER see Appendix A. The following security policies are well-formed formulae in first-order predicate calculus with equality, and notated in the syntax utilized by OTTER.

1. Rule 1

The sender of a classified document is obliged to update the document classification as soon as it is possible, i.e. immediately after the sender evaluates that the document classification is obsolete.

(all d

(Classified_Document(d) &

Transmitter(d) & Transmitter(d) = Agent(a) &
Res_Exec(Transmitter(d), Evaluate_Classification(d), level) &
Classification(d) != level

->

Obligated_To_Update_Classification(Transmitter(d),
Change_Classification(d)))
).

2. Rule 2

Any agent who is not the sender of a classified document is prohibited to change the classification of this document.

(all d

(Classified_Document(d) & Transmitter(d) != Agent(a)

->

Forbidden_To_Change_Classification(Transmitter(d),
Change_Classification(d))
)
).

3. Rule 3

The holder of a classified document is permitted to ask the sender to revise the classification of this document.

(all d

(Classified_Document(d) & Transmitter(d) & Holder(d)

->

Permitted_To_Ask_Revision(Holder(d),Ask(Transmitter(d),
Change_Classification(d)))

)

).

4. Rule 4

Every organization, which holds some secret documents, is obliged to designate an agent who is responsible for preserving these documents.

(all o (exists d (

Organization(o) & Classified_Document(d) & (Classification(d) = Secret)
& Works_For(Holder(d),Employees(o))

->

Obligated_To_Designate_Agent(

Head(o), Designate_Responsible(Document_Preservation(o)))

))

).

5. Rule 5

The sender of the classified document is obliged to establish a entrust note of this document. The head of the sending side is obliged to sign this entrust note. The head of the sender side is permitted to delegate the obligation to sign the note to one of his representatives.

(all o (exists d (

Organization(o) & Sending_Office(o) & Classified_Document(d)

->

Obligated_To_Establish_Note(Sending_Office(o),

Establish_Consignment_Note(d))

))

).

(all o (exists d (

Organization(o) & Sending_Office(o) & Classified_Document(d)

&

Decide_To_Send(o, Decide(Send(d))) & Consignment_Note(d)

->

Obligated_To_Sign_Note(Head(Sending_Office(o)),

Sign(Consignment_Note(d)))

))

).

(all o (exists d (

Organization(o) & Sending_Office(o) & Classified_Document(d)

&

Decide_To_Send(o, Decide(Send(d))) & Consignment_Note(d)

& Representative(d) &

Works_For(Representative(d),Head(Sending_Office(o)))

->

Permitted_To_Delegate_Sign(Head(Sending_Office(o)),

Delegate(Representative(d),Sign(Consignment_Note(d))))

))

).

6. Rule 6

Anybody body who wants to visit a restricted area is obliged to get an authorized from the head of that area. In addition to that, the visitor is obliged to be supervised by an agent who is specially designated for supervising visitors.

(Organization(o) & Protected_Area(a) & Head(a) = h & Person(p) &
(-(Work_For(p,o)) & -Exec(h, Authorize(p, Visit(a))))

->

Forbidden_To_Exec(p, Visit(a)).

(Organization(o) & Protected_Area(a) & Head(a) = h & Person(p) &
(-(Work_For(p,a))) & -Exec(h, Authorize(p, Visit(a))))

->

Forbidden_To_Exec(p, Visit(a)).

(Organization(o) & Protected_Area(a) & Head(a) = h & Person(p) &
-(Work_For(p,o)) & Exec(h, Authorize(p, Visit(a))) &
Exec(h, Designate_Responsible(Supervision(p))))

->

Permitted_To_Exec(p, Visit(a)).

7. Rule 7

The holder of the secret document and the witness of the destruction are obliged to sign destruction time at every secret level classified document destruction.

(all d

(Classified_Document(d) & (Classification(d) = Secret) & Holder(d)
&
DESTROY_DOCUMENT(d) & Actor(d) & Destroyed_Object(d)
&

Witness(d) & After(Destroyed_Object(d)) &
Destruction_Minutes(d)

->

Obligated_To_Sign_Destruction(Holder(d),
Sign(Destruction_Minutes(d)))
&Obligated_To_Sign_Destruction(Witness(d),
Sign(Destruction_Minutes(d)))
)

).

8. Rule 8

After each meeting, the organizer of the meeting is obliged to burn all the preparatory documents of this meeting completely.

(all d

(MEETING(d) & (Organizer(MEETING(d))) &
After(MEETING(d)) &

Element_Of_Meeting(d, Preparatory_Documents(MEETING(d)))

->

Obligated_To_Burn(Organizer(MEETING(d)), Incinerate(d))

)

).

9. Rule 9

The organizer of the meeting is obliged to keep these preparatory documents in a safe, unless the documents are burnt completely.

(all d

(MEETING(d) & (Organizer(MEETING(d))) &
After(MEETING(d)) &

Element_Of_MeetingDocuments(d,
Prepatory_Documents(MEETING(d)))

->

Obligated_To_Keep_In_Safe(Organizer(MEETING(d)),
(Element_Of(d, Contents(Safe(s)))))

)

).

10. Rule 10

The organizer of any meeting is obliged to establish a list of all participants before that meeting.

(all d

(MEETING(d) & (Organizer(MEETING(d))) &
Before(MEETING(d))

->

Obligated_To_Establish_List(Organizer(MEETING(d)),
Establish_Participants_List(MEETING(d)))

)

).

11. Rule 11

An agent is obliged to work a classified document at the secret level in a protected area.

(all d

(Agent(d) & Classified_Document(d) &

(Classification(d) = Secret) &

During_Exec(Agent(d), Elaborate(d))

->

```

        ( During_Exec(Agent(d), Work(Protected_Area(d))))
    )
).

```

12. Rule 12

After the destruction of a classified document at the secret level, the document holder is obliged to inform the author of the document.

```

(all d
    (Classified_Document(d) & (Classification(d) = Secret) &
    Holder(d) & Exec(Holder(d) , Destroy(d)) & (Transmitter(d))
    ->
    Obligated_To_Notify(Holder(d), Notify(Transmitter(d), Destroy(d)))
    )
).

```

B. REPRESENTING POLICY VIA UML DIAGRAMS

UML provides a collection of diagrams to capture different aspects of a system. For instance, use cases capture user requirements, class diagrams are used to capture the static nature of objects. Collaboration and sequence diagrams are used to capture dynamic interactions between objects and systems, and package and deployment diagrams organize design elements. Based on [8], we choose to represent policy via diagrams. Once the policy is represented in diagrams, test cases can be developed as discussed in the next section. The overall schema of the implementation is in Figure. 4.

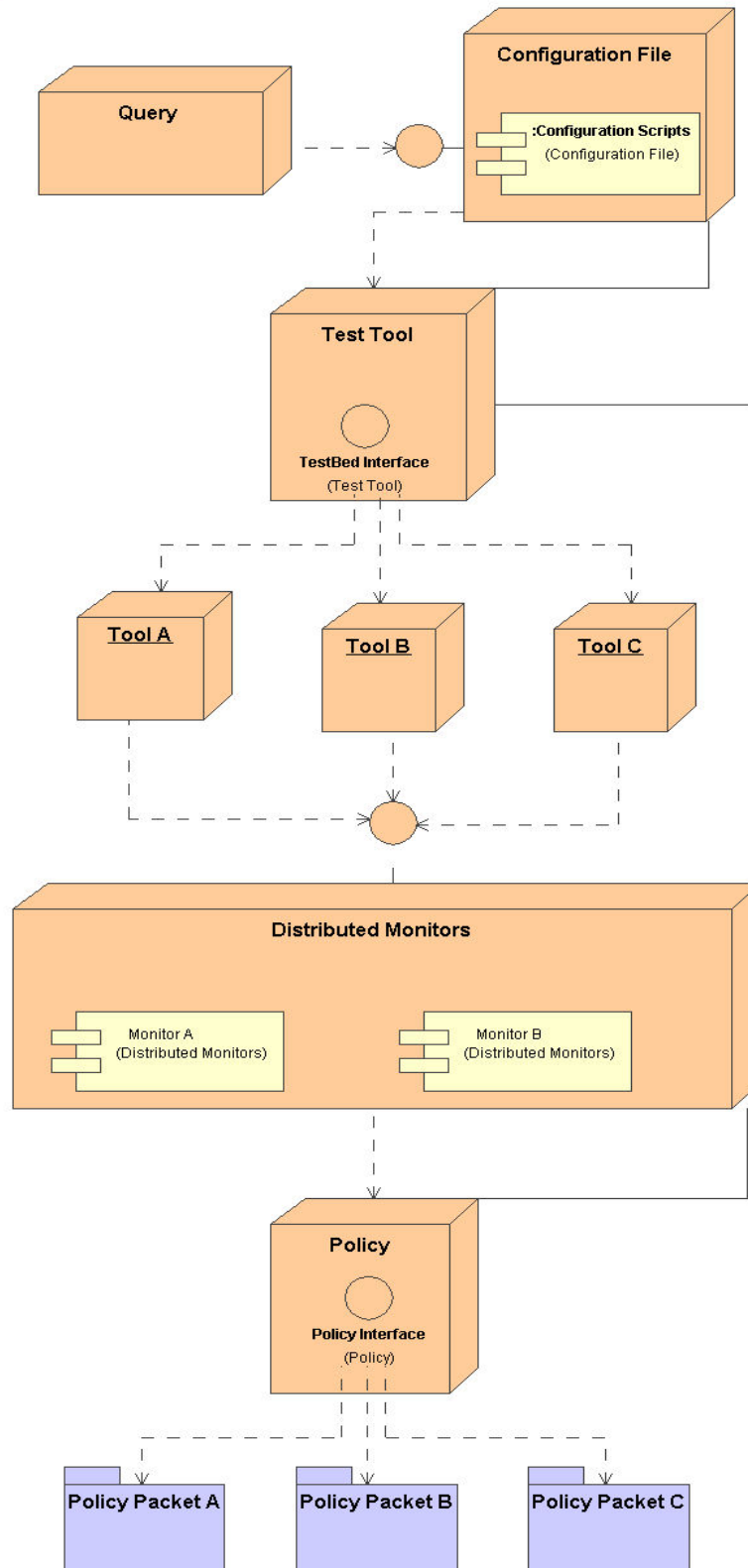


Figure 4. UML Diagram of the Overall Implementation Schema.

C. USING COLLABORATION DIAGRAMS FOR UML REPRESENTATION

Test criteria (i.e., a rule or collection of rules that impose test requirements on a set of test cases) can be set based on UML collaboration diagrams. Tests can be generated automatically from software design (i.e., policy design), rather than code or specifications. Criteria are defined for both static and dynamic testing of specification-level and instance-level collaboration diagrams.

Collaboration is a description of a collection of objects that interact to implement some behavior within a specific context. It contains slots (i.e., roles) that are filled by objects and links at run time [31]. A collaboration diagram is a graphical representation of collaboration. In an object-oriented computing paradigm, objects interact with each other to implement behavior. One way of representing this interaction is in terms of how they collaborate. In other words, collaboration diagrams show the interaction organized around objects that participate in the interaction and their links to each other. The objects in a collaboration diagram are instances of classes in a class diagram. A collaboration diagram has two forms [32]:

1. Specification-level Collaboration Diagrams

Specification-level collaboration diagrams show the roles defined within collaborations. The diagram contains a collection of class boxes and lines corresponding to ClassifierRole (i.e., a role to be played by an object within a collaboration) and AssociationRoles (i.e., a role that defines the relationship of a ClassifierRole to other roles) in the collaboration.

2. Instance-level Collaboration Diagrams

An instance-collaboration diagram shows the collaboration of the instances of ClassifierRoles and AssociationRoles.

Specification-level and instance-level collaboration diagrams describe the structural relationships among the participants of collaboration and their communication patterns. Collaboration diagrams describe the structure and

behavior of the system. These two forms of collaboration diagrams specify what requirements must be fulfilled by the objects in a system, and what communications must take place between the objects for a specific task to be performed. UML diagrams used at different levels of abstraction provide the following information [32]:

- The objects that are involved in an interaction and the structure of these objects.
- Instances of allowable sequences of operation calls to an object
- The semantics of an operation
- The operations that are imported from other classes, thus enabling a collaboration with objects of the other class
- The communication pattern of objects in a collaboration (synchronous or asynchronous)
- The execution characteristics of objects (parallel or sequential)

Using this information, it is possible to generate tests from collaboration diagrams. In this thesis, collaboration diagrams are used to structure policy and therefore facilitate testing policy.

Testing can be either static or dynamic [32]. Static testing involves checking some aspects of the policy specification without execution. For the static testing aspect, collaboration diagrams provide four items to statistically check the SUT:

- Classifier Roles
- Collaborating Pairs
- Message or Stimulus
- Local Variable Definition-Usage Link Pairs

Dynamic testing involves testing based on inputs. For the purposes of dynamic testing, collaboration diagrams provide the realization of operations. Each collaboration diagram represents a complete set of messages during the execution of a rule. Therefore, a complete set of message sequence path can be traced. The realization of the operations during the execution of that rule can be tracked.

D. WHAT ARE THE TEST CASES GIVEN THESE RULES?

Based on the given policy rules, test cases are required to be developed for policy testing. Depending on the rule, the setup of the test case differs. Each rule needs test cases to be tailored accordingly. Defining the actions and actors plays a crucial role for tailoring the test cases to the policy. However there is not necessarily a one-to-one correspondence between the actions in policy testing and those of the actual software implementation of the actions. The software implementations of the actions reside at a finer level of abstraction. In contrast, the actions in policy testing are modeled at a course level of granularity. To develop a fruitful set of test cases the considerations that guide test-case generation is discussed for each of the rules in the case study.

1. Rule 1

In this rule, the user (i.e., sender) is required to:

- Evaluate the classification of the classified document.
- Update the classification of the document if the classification for that document is obsolete.

The sender of the classified document is obliged to update the classification level of the document as soon as the sender evaluates that the classification of the document is obsolete. “As soon as” needs to be clarified in the policy to ensure timely updates by the policy users. There is some degree of

ambiguity about the order of the actions. Before proceeding to testing, the policy workbench must either:

- Ask the user for clarification, or
- Try to make an intelligence guess at the correct interpretation.

Asking the policy user user for clarification encourages the generation of feedback that can be used to tailor the policy according to a particular scenario. However, such an approach requires the user to know about the temporal property(i.e., the time when a classified document becomes obsolete). The interpretation of the temporal property should not be left to system, because the user might interpret it as a choice left to the user which will become an obstacle in policy enforcement. Attempting to make an intelligence guess will facilitate the policy testing and does not depend on user input. However, a database of previous temporal properties associated with this rule needs to be stored. Thus, the policy workbench tool can invoke the temporal property database and based on the result set, makes an intelligent guess. A sequence of events takes place before the policy user's evaluation of the classification level. At first, it must be assured that the user gets a document. Otherwise, it is out of the scope of this policy.

2. Rule 2

Any agent in the organization is required to do the following:

- Check the sender of the classified document
- If that particular agent is not the sender, then the agent is prohibited to change the classification. At this point, the test is checking against a Boolean value `Test (Role = Sender) -> {True, False}`.

This rule does not enforce either a temporal property or a sequence property.

3. Rule 3

One of the actors in this rule, the sender is required to send the document. The holder (i.e., the receiver of the document) is required to:

- Evaluate the classification of the document
- Notify the sender to revise the classification of the document if needed

This rule also does not enforce a temporal property. But the sequence of the events needs to be tested. First of all, the receiver receives a classified document from the sender, and determines that the classification of the document needs to be revised. The receiver next requests a revision from the sender. As it is stated in the rule that only the holder can ask for revision, the counting property needs to be checked to make sure that there is a sender and a holder.

4. Rule 4

The holder (i.e., the department head, or section head) is required to:

- Evaluate the classification level of the classified documents
- If secret, assign an agent to preserve the documents

The agent is responsible for preserving the documents labeled Secret. As a counting property, if there is a secret document, then there must be a document holder to assign an agent and an agent to preserve the document. The number of actors in each role type must be counted, because the policy may specify any arbitrarily large number of actors in each role type. The responsibility of the agent starts when the holder of the document fulfills the obligation of evaluating the classification of a document labeled Secret and designating an agent(Precondition).

5. Rule 5

The sender (i.e., the agent who is required to prepare the classified document) is obliged to:

- Prepare the classified document
- Establish an entrust note
- Pass the document to the head of the department for signature

The department head of the sender is obliged to sign this entrust note. If the head of the department delegates its signing authority, the delegated representative is obliged to sign the entrust note. In this rule, there must be a sender and an authority to sign the entrust note. The signing authority can either be the head of the department or the representative.

6. Rule 6

The visitor is required to:

- Get an authorization from the head of that restricted area
- If the visitor gets the authorization to enter, then the visitor waits to be supervised by a supervisor who is assigned

The counting property needs to be tested are the visitor, the head of the restricted area, and supervising agent. The sequence of the events needs to be fulfilled before visiting the restricted area are:

- Obtain authorization
- Assign a supervisor
- Visit the area

7. Rule 7

The holder of the classified document required doing the following for destruction of every document labeled Secret:

- Find an agent (i.e., witness) for the destruction
- Destroy the document

- Sign destruction time, date, and name of holder of the document

The witness is required to sign the destruction time, date and name of the witness after destruction. The temporal property needs to be tested is the destruction time and date. The holder agent and witness must be present at the time of destruction. Therefore, the counting property needs to be used to test for the existence of two actors. In the sequence of events, after every destruction the signing must take place.

8. Rule 8

The organizer is obliged to burn the preparatory documents of the meeting completely. In the sequence of events, the organizer must burn the preparatory documents after the meeting is held. The only actor in this rule is the organizer.

9. Rule 9

Unless the organizer of the meeting burns preparatory documents, the organizer is required to keep preparatory documents in a safe. In other words, the testing has a if-then-else pattern in this rule.

10. Rule 10

For each meeting, the organizer who is the only actor in this rule is required to establish a participants' list. As a temporal property, the list must be prepared before the meeting. Therefore, the due time of the participants' list must be checked to be before the starting of the meeting.

11. Rule 11

The agent is required to work on the secret-level classified document in a protected area. Temporal test cases must test the time the agent works on a Secret_level document and whether it is in a protected area or not. The only actor in this rule is the agent.

“What would happen if the agent moves to another work area in which it has authorization to enter?” To test such a scenario, based on the background

knowledge or the policies in place, the agent might be searched. If the search results indicate that the agent is mobile, then the agent is given the right to work with the secret document in the new protected area. The mobility can also be confined to either to a specific policy environment or a broader environment. The testing needs to test whether the agent's move is within permitted boundaries. For example, Agent_A has full mobility in places where policy objects P_A, P_B are valid. If the Agent_A's movement is within P_A and P_B's boundaries then permission is granted. Otherwise, it is prohibited.

12. Rule12

The author and the holder of the document are the two actors in this rule (counting property). The holder of the document is required to do following:

- Destroy secret level document
- Sign destruction time and date
- Inform the author of the secret level document

Test cases are required to test the sequence of events that the holder of the document labeled Secret does during destruction of the document and notification of the author.

Test Property Rule #	Temporal Property	Counting Property	Sequence Property	Combination
Rule 1	1	0	1	101
Rule 2	0	1	0	010
Rule 3	0	1	1	011
Rule 4	0	1	1	011+ Precondition
Rule 5	0	1	1	011
Rule 6	0	1	1	011
Rule 7	1	1	1	111
Rule 8	1	0	1	101
Rule 9	0	1	1	011
Rule 10	1	1	1	111
Rule 11	1	1	0	110+ Scenario
Rule 12	0	1	1	011

Table 1. Policy Test Property Distribution Table.(1= applies to that rule, 0=does not apply to that rule).

E. HOW CAN WE HANDLE DIFFERENT QUERIES AT THE TEST SPECTRUM?

The test spectrum has query-specific tests at one end, and the generic types of tests at the other end. In the middle of this spectrum are various combinations of these types. Different kinds of queries are created to test the policy base across the spectrum. The policy base is dynamic, so that the tactical testing strategy needs to be changeable on-the-fly.

Picking up a random set of test cases and trying to test the policy subset or policy subsets will force exhaustive testing which is known to be computationally intractable. In the worst case, we need to test all of the policy subsets and the interactions among them. Under best-case conditions, testing representative parts of policy subsets and their relations is sufficient.

Commonly tested features in the rules of policy subsets can be gathered into generic test types. The key effort is to design the policy base in an object-oriented way such that it eases the creation of generic test types. Even in already-built policies, the testers can give the feedback to the policy makers to state the policy in a different way that facilitates the creation of generic types. The policy makers should ask the following questions:

- Are the rules in different policy subsets equivalent or can they be stated in a different way?
- How can we state the policy rules in such a way that makes it readable by computer?

Policy subsets are not stand alone objects in the organizations, they interact with each other. The generic types can be used to test different policy subsets working in the same environment. Testing policy subsets separately does not give inconsistencies that emerge when policy subsets tested together.

The automatic testing tool is takes the policies that are involved in the testing in the generic query,stepping through the entire policy base under test

(PBUT) or policy base subset under test(PBSUT). At this point, the tool tries to decide:

- What combination does each policy subset fall under?

For each combination, the system will do summation. For example, fifteen of the '101' patterns, five of the '001' patterns, and so on. Even in our case study, which consists of a small policy base, patterns emerge. Then by using expert system, the system decides in this sequence:

The PBUT has fifteen of the '101' pattern, five of the '001' pattern and so on, see Table 1.

What do these results mean in terms of the computational of the test cases to be run. The system can be designed to have heuristic rules that guide the automatic test tool at this point. An example of heuristic rules can be:

- Telling the user that the testing process is going to take a long time, does he/she want to continue testing?
- Telling the user to refine the query that is submitted. Otherwise, the system is going to perform intensive testing.
- Ask the policy workbench user to weight the importance of each of the tests or types of test cases

The heuristic rules guide the system to user-defined paths. The advantage of that is that the user is informed of the sequence of the events that will take place. Unless the user is informed of the events, at the end the system might end up with an infinite number of candidate tests to run or come up with nothing practical. For practicality reasons, the user is given the chance to guide the test-case selection process at critical points. Most of the cases the testing results closed to a reference-testing curve is enough and practical. Thus heuristic rules enable timely and practical responses. However, it is unclear how practical it is to rely on the policy user to make wise decisions in this regard.

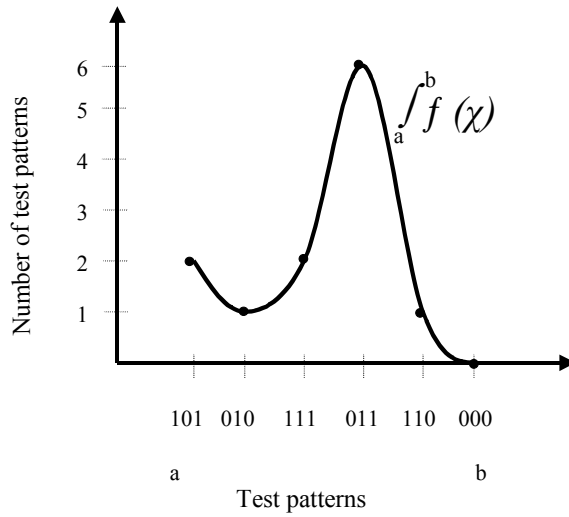


Figure 4. Relative Frequency Distribution of the Test Cases for the Policy Base in the Case Study.

The histogram of the test patterns/cases shows the function of the test patterns under PBUTs. The shape of the curve gives us what strategy to use in testing that SUT and where to focus our testing effort to obtain the most fruitful results with a minimum of the testing resources.

The system keeps the history for each of the PBUT tests. Given that particular types of testing patterns that are executed (successfully or not), statistical inference can be done as described below:

The policy-workbench user applied the patterns after running n number of test cases, the meta statistics say that the system has thirty-five percent that will succeed under given a set of PBUTs. Under these conditions, “what group of patterns has the highest probability of being within the correct testing area?” Based on the factors such as resources at hand, time constraint and criticality of the testing, the testers decided to execute all of the or some of the picked-up patterns. One of the advantages of visualizing the overall pattern of the PBUT is

that the tester has the big picture which guides him to create his testing plan(i.e., the road map for the testing process). Thus, a more realistic schema-based testing is available. The second advantage is to let the tester map between the highly probable patterns and criticality of these patterns. In Figure 4, the pattern ‘011’ is three times likely than any of the other patterns. Thus, the testing tool can check this highly probable pattern’s relative importance and focus on it accordingly. For instance, ‘000’ pattern does not have any of the properties. But it still conveys a meaning to policy-workbench user. It represents a policy that only offers guidance or states facts.

Using a rule pattern language facilitates pattern recognition effort in large policy bases. Arsanjani[53] proposed a rule pattern language that can be used for expressing patterns.

F. QUERY MAPPING

The PBUT has rules(i.e., $R_1(o_1, o_2, o_3, \dots, o_n)$) and the testers develop queries such as $Q_1(t_1, t_2, t_3, \dots, t_n)$ to test the PBUT. The mapping procedure is as follows:

- Map terms in queries such as $t_1, t_2, t_3, \dots, t_n$ to each rule in the PBUT such as $o_1, o_2, o_3, \dots, o_n$ (Figure 5).
- Map terms o’s such as $o_1, o_2, o_3 \dots o_n$ in rules to the o’s in other rules. For instance search for the mapping between the o’s in R_1 and R_3 (**Inter-queryrelationship**). See Figure 6.
- Map terms o’s such as $o_3 \dots o_n$ in each rule to the other o’s in the same rule. For instance search for mapping between the o’s in R_1 (**Intra relationship**). See Figure 7.
- Search for predicates such as $P(P_1, P_2)$, which is between the predicates (i.e., $P_1(o_1, o_2), P_2(o_2, o_3)$) that are found in earlier steps (**Semantic mapping**). See Figure 8.

- Search for semantic mapping among queries such as $P_1(Q_1, Q_2)$.

Firstly, the system tries to search for mapping from the terms of the queries to the related rule terms. Searching for such a mapping enables us to figure out the rules involved in that particular query and which parts of the rules are involved in it. Each of the queries are executed in this mapping process.

Secondly, mapping among the terms of the rules in different rules are searched. Once the mappings of the terms among PBUT rules is completed, then the interdependency and relation types of the rules among them can be easily shown. For instance, t_2 of Q_1 can have relationships to o_1, o_3 of R_1, o_2, o_3 of R_3 . Even if Q_1 does not directly has a relationship to the o_4 of R_3 , because o_3 of R_1 has relationship to o_4 of R_3 , Q_1 has an indirect relationship to o_4 of R_3 . If the inter relationship among the rules are not searched, this dependency does not come up and is skipped by the testing process.

Thirdly, the mapping among the terms of the same rule is searched. The terms of the rules do not necessarily have to be variables: they can also be predicates. Thus, there might be a relationship among them. Just testing the query terms that directly relate to the terms or predicates of the relevant rule might cause the testing tool to skip testing the relationship among rule terms. For instance, t_2 of Q_1 can have relationships to o_1, o_3 of R_1 . The o_3 of R_1 has intra relationship to o_5 of same rule which is R_1 . Even if t_2 of Q_1 does not directly have a relationship to o_5 of R_1 , there exists an indirect relationship among them. If the tester only tests direct relationships, then the semantic relationship might be skipped. Having a detailed mapping of terms facilitates the detected of such as gaps, and inconsistencies in the PBUT.

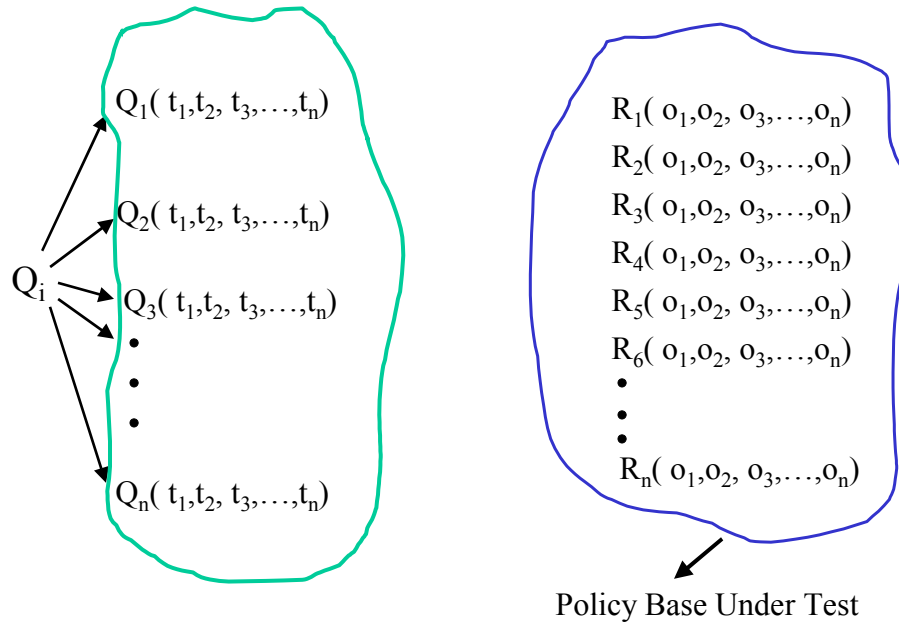


Figure 5. Query to Rules Mapping Diagram.

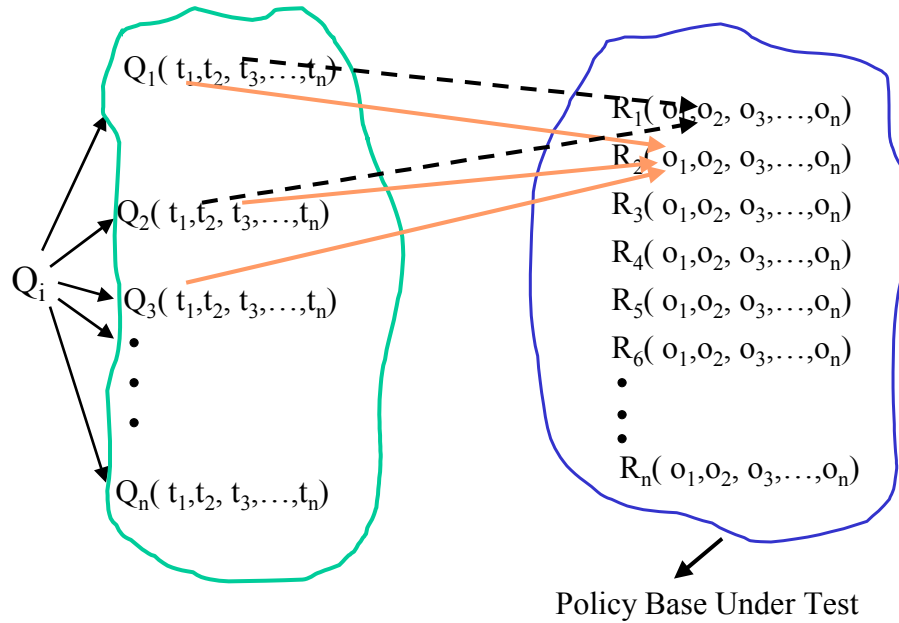


Figure 6. Inter-query Relationship Diagram.

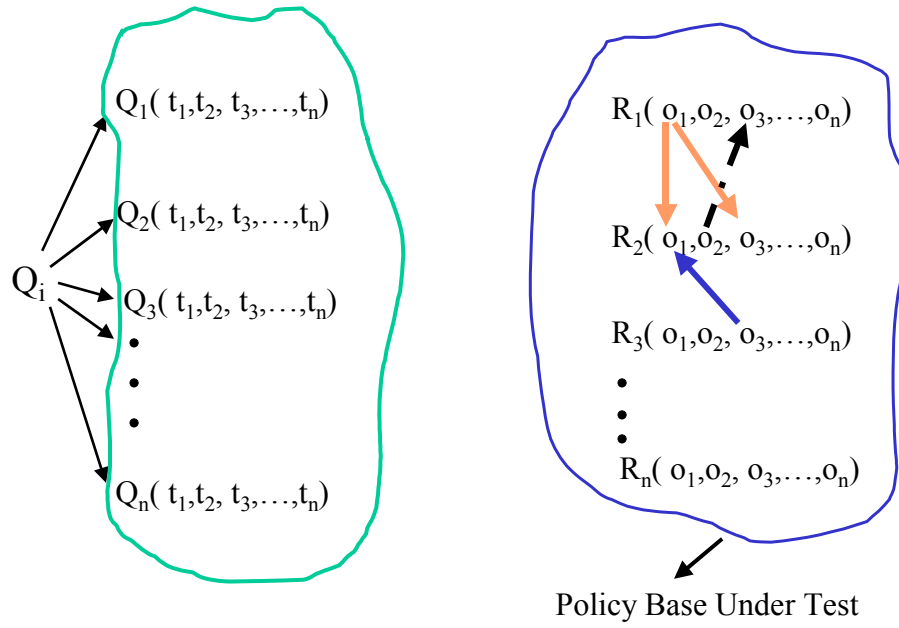


Figure 7. Intra-Relationship Diagram.

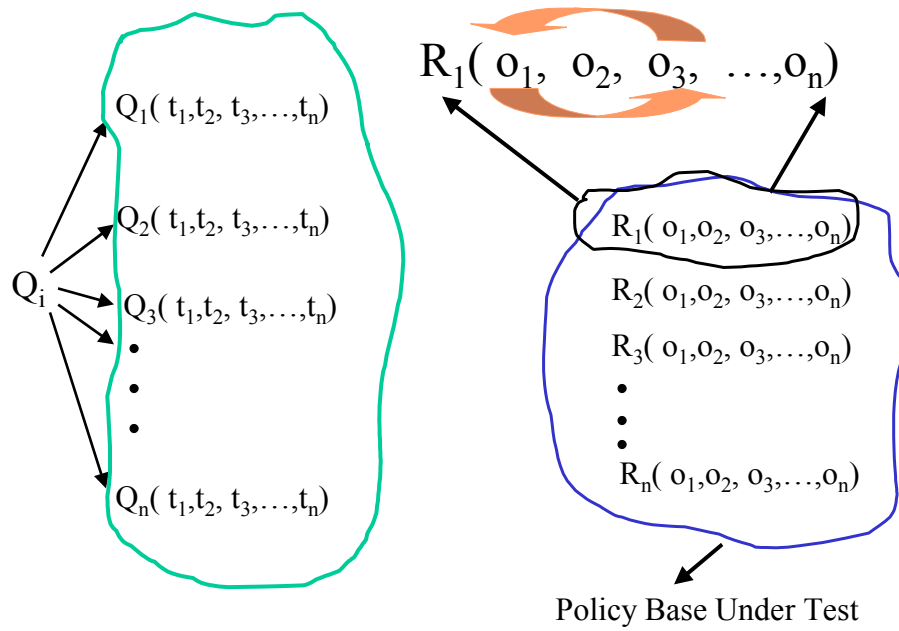


Figure 8. Semantic Mapping Diagram.

G. FUZZY LOGIC AND POLICY TESTING

Unquestionably, computers proved to be highly effective in dealing with mechanistic systems, that is inanimate systems whose behavior is governed by the laws of physics, chemistry, and electromagnetism. Unfortunately, the same cannot be said about humanistic systems[48].

Zadeh proposed to abandon the high standards of rigor and precision that we have become accustomed to expect and become more tolerant of approaches which are approximate in nature. The problem is the excessively wide gap between the precision of logic and the imprecision of the real world. Policy-making is a human enterprise. As such it integrates many complementary, contradictory, fuzzy, and changing human values[46].

Fuzzy logic, in a narrow sense, is a logical system that aims at a formalization of approximate reasoning[47]. In a broad sense, Zadeh asserted that fuzzy logic is almost synonymous with fuzzy set theory. Fuzzy set theory is basically a theory of classes with unsharp boundaries, that is a class in which the transition from membership to non-membership is gradual rather than abrupt[48]. In short, fuzzy logic extends traditional logic and enables it to handle approximations and nonnumeric variables.

1. What does Fuzzy Logic Offer?

Fuzzy logic provides a systematic way to deal with reasoning under uncertainty. There exist real-life situations to represent and reason about uncertainty for which probability and randomness do not suffice. Zimmerman [49] states that fuzzy logic offers the rigor of formal methods without requiring undue precision. It also offers alternative methods to handle policy preferences and conflicts.

Despite fuzzy logic work[47] on modeling computer security, policy using fuzzy logic little else of substance has been reported on the use of fuzzy logic in policy structuring and testing.

2. What can be the Fuzzy Concepts in the Policy Workbench?

- Actors or agents
- Objects
- Rules
- Risk
- Informal policy rules
- Natural language policy objectives
- Natural language policy specifications
- Policy testing techniques
- Classification levels

Prototypes[51] are used as a useful notation for simplifying the complexity of the real world. The concepts mentioned above can be considered to be fuzzy logic prototypes for a policy workbench. Different classes of policies can be also considered as prototypes. These prototypes enable one to maintain a simple and useful concept without the diversity and complexity of the members of these fuzzy sets. The test-patterns concept discussed before fits into the prototypes. The test patterns can facilitate the tester's conversion to prototypes.

3. What Techniques of Fuzzy Logic can be Applied to Policy Testing?

Fuzzy logic techniques that can be applied to policy structuring and testing can include the following: fuzzy constraints, realistic policy modeling, nonnumeric variables, and fuzzy decision-making.

A fuzzy constraint is an objective, that can be characterized as a fuzzy set in an appropriate space [49]. For instance, an example of a fuzzy constraint within

our case study is “The organizer of the meeting is obliged to incinerate the preparatory documents of the meeting within twenty-four hours after the meeting.” By converting the existing rules slightly, they can have the form which we can apply fuzzy logic. Once the rules in the policy are put into a fuzzy form, one can use fuzzy logic to reason about “short” vs. “long” vs. “sort of short” etc. The fuzzy counterpart of the same rule is “The organizer of the meeting is obliged to incinerate the preparatory documents of the meeting a short time after the meeting.” Therefore, the fuzzy constraint is “Burn_The_Preparatory_Documents in a short time.”

A fuzzy goal is an objective which can be characterized as a fuzzy set in an appropriate space [49]. For instance, “The sender of a classified document is obliged to update the document classification as soon as it is possible, that is, immediately after the sender evaluates that the document classification is obsolete” is a typical goal. Based on this policy, some examples of the fuzzy goals can be as follows:

- Update immediately
- Update after evaluation

A fuzzy decision is the fuzzy set of alternatives resulting from the confluence of the goals and constraints [49]. As an example, the fuzzy decisions can be expressed as the intersection of the fuzzy goal and constraint sets.

H. ANYTIME REASONING IN POLICIES

First-order logic (see for example [42]) is used to structure and test policy in this thesis. In addition to our work, first-order logic is used in a variety of applications such as verification of hardware, and software, and solving resource allocation and scheduling problems [41]. However, the use of first-order logic automated reasoning in applications with real-time decision-making constraints has been inhibited because of the inability to provide good estimates of the bounds on the resources needed to solve even very small problems [39]. However,

some applications require that the best decision possible with limited resources for reasoning (usually the limited resource is time) be made. Anytime reasoning algorithms provide weaker or stronger approximations of the satisfiable sentences within practical usage of resources. Algorithms which give best-effort solutions to problems given bounds on the resources available for reasoning are known as anytime reasoning algorithms [39]. Anytime reasoning addresses the problem of intractability in first-order logic reasoning problems.

Anytime reasoning uses algorithms whose accuracy of results improve gradually as computation time increases, providing a tradeoff between resource consumption and output quality [43].

In testing policy, the end user should have the ability to make a choice between a practical answer and an answer that will take too long to generate to be of practical use. Organizations keep the testing results in a database. Before the automatic testing tool processes the user query, it searches the query database and retrieves previously executed queries and their results related to this query. The sequence of events is as follows:

- Search for previous queries
- Assess whether the PB has changed
 - Check the policies involved in the query
 - Perform dependency analysis(i.e., traceability)

After this retrieval, if the testing tool searches for a contradiction, it checks for incompleteness and gaps in the policy and estimates whether the test will take a longer time. The tool asks the user whether it should continue the long testing process or use an anytime reasoning algorithm that will provide approximations of the real testing results. The user needs to be able to override the advice of the tool. For example, the user may be very risk averse and want a very thorough test of the PBUT. If the user selects the latter approach, the system can also inform the end user whether the approximations are weaker or stronger than the expected test

results. Thus, the end user has an idea about the confidence interval of the results. But the end user does not necessarily search for an exact answer consisting of tight bounds; wider bounds defining a larger interval might fulfill end user requirements expected from testing.

The system can also have a number of anytime algorithms in its database. Using different anytime algorithms can give healthier approximation results. For different anytime algorithms, see [43], [44], [45]. Using different anytime reasoning algorithm can tailor different types of policy-object querying.

I. TARGET TEST PATTERN SELECTION

The information repository of the test patterns are comprised of all of the test patterns related to PBUTs. The normal distribution of the patterns can be developed from the repository. Most of the testing efforts does not permit exhaustive testing. In theory, the testing tool could test all of the patterns in the repository. However in reality, the tools must rely on performing sampling. The sampling process brings an endpoint to the testing tool for running test cases.

Actually, if the PBUT is small enough, the testing team can run all the test cases in the repository. However, this is not the typical case. If the PBUT is not small enough to run all the test cases(i.e., N), the testing team picks up a target set of test case region within that repository(Figure 9). After the selection of the target region, the testing team checks whether the working region and the resources at hand is small enough to test all test patterns in the target region. If so, the testing tool will run test patterns. Otherwise, the tool will pick up only a sample of the patterns(i.e., n) in the target region(Figure 10).

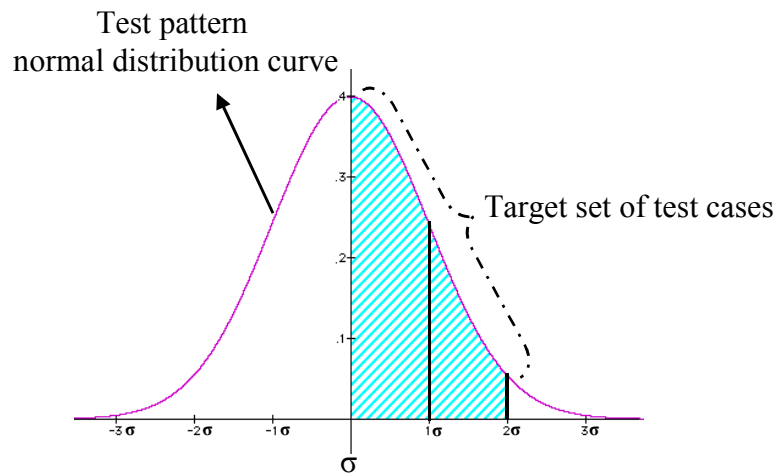


Figure 9. Target Test Pattern Region Selection Diagram.

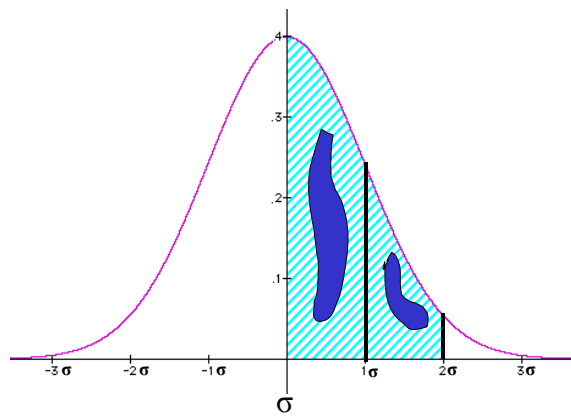


Figure 10. Sampling Diagram within Selected Target Regions Diagram.

The sampling within the target region associated with factors such as loss of productivity, loss of property or damage, and injury or loss of human life stem from untested areas. Tersely, risk is associated with the sampling process. For instance, if the PBUT is a DOD system, this risk might result in loss of mission, loss of secrets, or operational losses. If the PBUT is from industry, loss of prestige, loss of revenues associated with anti-lawsuits, or loss of productivity could result. The risk can defined as follows:

$$\text{Risk} = f \left(\frac{\text{\# of successfully run test cases}}{\text{Total \# of test cases within that target region}} \right)$$

vs.

Magnitude and frequency of losses from exercising the sample of target region with losses due to gaps in the PBUT.

J. EXAMPLE OF TARGET PATTERN SELECTION

Let's take a PBUT whose quality of the policy base is unknown. The target pattern selection process can be as follows:

- Create a baseline(i.e., B_0) from the first round of testing.
- Based on phase I, if we detected one of the following:
 - The presence of one or more gaps is detected during operation of the PBUT.
 - We update the PBUT.
- Run another round of tests

Either one of the following outcomes can occur. The testing tool can run another round of tests. They do not necessarily run the same tests. If the tool detects gaps, a new set of patterns to elaborate on gaps will need to be generated. If an update occurs, the PBUT is modified, and it becomes necessary to recompute the test pattern distribution curve and regenerate test cases.

- Create a new version of B_0

The previous versions of B_0 gives this pattern selection a roll-back capability, as part of a configuration management capability. In the testing of policy, there is also a need for configuration control. Considering software engineering change notices; the head of the testing team can decide either accept, reject, or defer an update request.

K. EXECUTION OF THE IMPLEMENTATION

Resolution-style theorem prover is used in this thesis. The workflow of the proofs is in Figure 11. Firstly, the theorem prover engine tries to find a proof for a given Set of Support(SOS). If SOS for a given test case, is too large then the time-to-proof may be long. If no proof is generated, then the SOS must be revised in this case to limit the search space. In contrast, if SOS is too small, then the theorem prover may have too few axioms with which to generate a proof. Thus, the test tool will need to use such feedback information to dynamically update the SOS for testing purposes. However, in either case – too large or small of an SOS – one does not know whether the proof process was unsuccessful due to the size of SOS or because of the incompleteness of the policy base itself.

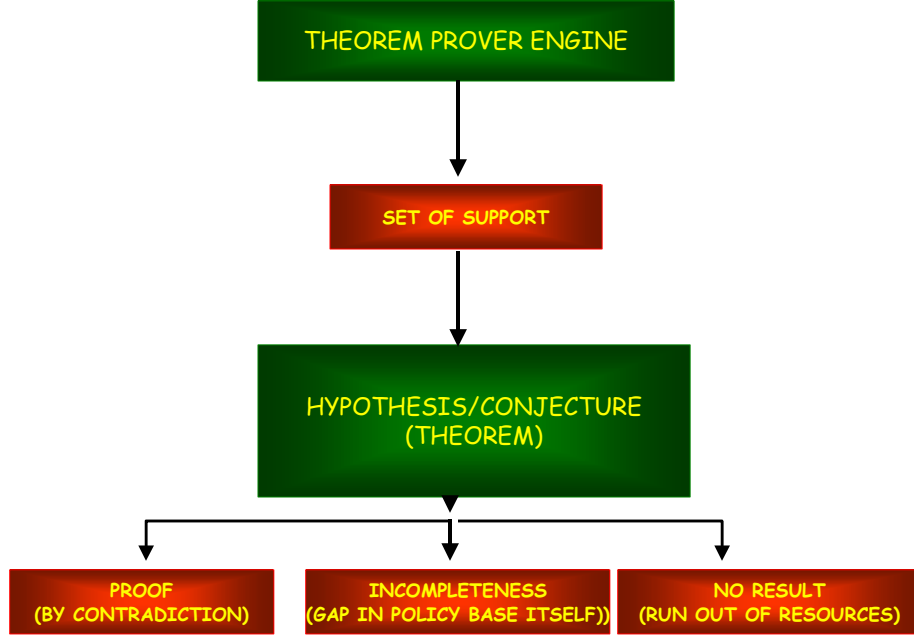


Figure 11. Resolution-style Theorem Prover Workflow Diagram.

We run the queries under the following environment:

- The query in natural language form is converted into first-order logic form before the search for a proof.
- The proof methods used in this thesis is proof by contradiction(i.e., refutation)
- The parameter controlling the maximum number proofs that are generated is set to 2.

The patterns of in this policy base distribute as in Table 1. Using these patterns we create a test suite associated with user queries or proposed updates. As we develop and group test patterns in the policy base, we can use patterns to create test suites that guides theorem-prover. We do sampling within selected target patterns. Then, we can develop queries that focus on each target pattern: The Query 1 tests ‘010’ pattern, Query 2 tests ‘011’ pattern, Query 3tests ‘110’ pattern, Query 4 does not test a specific pattern, Query 5 tests ‘111’ pattern,

Query 6 tests ‘011’ pattern. This test suite addresses all of the target patterns(five of the test patterns out of six) for this policy base and does not concentrate testing on a particular pattern. Moreover, Query 4 introduces an outlier query for the test suite.

For longer proofs, only one of the proofs is included in the body of thesis. Others can be found in Appendix.

1. Query 1

Query 1 in natural language(Q_{NL}):

“Are Representative, Holder, Witness, Agent or Organizer of the meetings employees of the organization?”

Q_{NL} is converted into Q_{COMP} , to test the query in theorem prover. The Q_{COMP} of this query is:

```

-(exists d (
    (Representative(d) | Holder(d) |
    Witness(d) | Agent(d)|Organizer(MEETING(d)))
    ->
    Employees(o)
)
).
```

The query executed in OTTER, the result of the proof in a concise form is(for a detailed output of the proof see Appendix D:

----- PROOF -----

19 [] -Agent(x16)|Works_For(Agent(x16),Employees(o)).

59 [] Agent(x26).

60 [] -Works_For(Agent(x26),Employees(o)).

69 [hyper,19,59] Works_For(Agent(x),Employees(o)).

70 [binary,69.1,60.1] \$F.

----- end of proof -----

The OTTER comes to a proof by using the clauses of 19 and 59(i.e. - Agent(x16) | Works_For(Agent(x16),Employees(o)) and Agent(x26)). The OTTER reached to proof at one proof level by hyperresolution.

2. Query 2

Query 2 in Q_{NL}:

“As a head of the sending office, am I required to sign the consignment note attached to the classified document originating from our sending office?”

Q_{NL} is converted into Q_{COMP}, to test the query in theorem prover. The Q_{COMP} of this query is:

-(exists o (exists d (

Organization(o) & Sending_Office(o) & Classified_Document(d) &

Decide_To_Send(o, Decide(Send(d))) & Consignment_Note(d)

->

Obligated_To_Sign_Note(Head(Sending_Office(o)),

Sign(Consignment_Note(d)))

))

).

The query is executed in OTTER, the result of the proof in a concise form is:

----- PROOF -----

6 [] -Organization(x6)| -Sending_Office(x6)| -Classified_Document(\$f3(x6))|

-Decide_To_Send(x6,Decide(Send(\$f3(x6))))|

-Consignment_Note(\$f3(x6))|

Obligated_To_Sign_Note(Head(Sending_Office(x6)),

Sign(Consignment_Note(\$f3(x6)))).

59 [] Organization(x26).

60 [] Sending_Office(x26).

61 [] Classified_Document(x27).

62 [] Decide_To_Send(x26,Decide(Send(x27))).

63 [] Consignment_Note(x27).

64[]-Obligated_To_Sign_Note(Head(Sending_Office(x26)),

Sign(Consignment_Note(x27))).

131[hyper,6,59,60,61,62,63] Obligated_To_Sign_Note(Head(Sending_Office(x)),

Sign(Consignment_Note(\$f3(x)))).

132 [binary,131.1,64.1] \$F.

----- end of proof -----

The OTTER comes to a proof by using the clauses of 6,59,60,61,62 and 63. The OTTER uses hyperresolution and binary resolution.

3. Query 3

Query 3 in Q_{NL} :

“While the agent is working on a classified document at secret level, does she required to elaborate that document in a protected area, or can she work in a normal office space?”

Q_{NL} is converted into Q_{COMP} , to test the query in theorem prover. The Q_{COMP} of this query is:

-(exists d

(Agent(d) & Classified_Document(d) & (Classification(d) = Secret) &

During_Exec(Agent(d), Elaborate(d))

```

->
( During_Exec(Agent(d), Work(Protected_Area(d))))
)
).

```

The query is executed in OTTER, the result of the proof in a concise form is:

----- PROOF -----

```

15 [] -Agent(x12)| -Classified_Document(x12)|Classification(x12)!=Secret |
-During_Exec(Agent(x12),Elaborate(x12))|

```

```

During_Exec(Agent(x12),Work(Protected_Area(x12))).

```

```

59 [] Agent(x26).

```

```

60 [] Classified_Document(x26).

```

```

61 [] Classification(x26)=Secret.

```

```

62 [] During_Exec(Agent(x26),Elaborate(x26)).

```

```

63 [] -During_Exec(Agent(x26),Work(Protected_Area(x26))).

```

```

88 [hyper,15,59,60,61,62] During_Exec(Agent(x),Work(Protected_Area(x))).

```

```

89 [binary,88.1,63.1] $F.

```

----- end of proof -----

The OTTER comes to a proof by using the clauses of 15,59,60,61, and 62. The OTTER uses hyperresolution and binary resolution. This query takes 11 miliseconds more than the Q₁ to reach to a proof.

4. Query 4

Query 4 in Q_{NL}:

“What happens if the classification level evaluated is same with the current classification level?”

The contradiction is searched in this query is “There is not any document whose current level of classification is same with the evaluated classification.” Q_{NL} is converted into Q_{COMP} , to test the query in theorem prover. The Q_{COMP} of this query is:

```

-(exists d
  (Classified_Document(d)
    ->
    -(exists level (Evaluate_Classification(d) != level
      )))
).
```

The OTTER does not reach to a proof in this query. This query is run to the maximum of 100000 seconds, 5091 clauses are kept for search. 38 hyperresolutions are generated, 3857 paramodulations are generated. The size of the SOS is 4256. The result of this query are:

- Improve the generation of test cases to get a proof in short time.
- Try to optimize the SOS by adding more additional real-world facts to help the OTTER reach a proof in short time.
- The systematic way of creating of test cases proposed in this thesis does not necessarily guarantee proofs within short time limits. The gap must be filled with optimization of the test cases based upon tests.

5. Query 5

Query 5 in Q_{NL} :

“Two agents working in the same department of the organization. One of them takes the other as a witness to the destruction of the classified document at secret level. Does the witness has to sign the destruction minutes?”

The contradiction is searched in this query is “There is not any witness who is obliged to sign the destruction minutes of the classified document that is destructed.” Q_{NL} is converted into Q_{COMP} , to test the query in theorem prover. The Q_{COMP} of this query is:

```

-(exists d
  (Classified_Document(d) & (Classification(d) = Secret) & Holder(d)
    &
    DESTROY_DOCUMENT(d) & Actor(d) & Destroyed_Object(d) &
    Witness(d) & After(Destroyed_Object(d))
    & Destruction_Minutes(d)
    ->
    Obligated_To_Sign_Destruction(Witness(d),
    Sign(Destruction_Minutes(d)))
  )
).
```

The query is executed in OTTER, the result of the proof in a concise form is:

----- PROOF -----

```

11 [] -Classified_Document(x8)|Classification(x8)!=Secret| -Holder(x8)| -
DESTROY_DOCUMENT(x8)| -Actor(x8)| -Destroyed_Object(x8)| -Witness(x8)|
-After(Destroyed_Object(x8))| -
Destruction_Minutes(x8)|Obligated_To_Sign_Destruction(Witness(x8),
Sign(Destruction_Minutes(x8))).
59 [] Classified_Document(x26).
60 [] Classification(x26)=Secret.
61 [] Holder(x26).
```

62 [] DESTROY_DOCUMENT(x26).

63 [] Actor(x26).

64 [] Destroyed_Object(x26).

65 [] Witness(x26).

66 [] After(Destroyed_Object(x26)).

67 [] Destruction_Minutes(x26).

68[]-Obligated_To_Sign_Destruction(Witness(x26),
Sign(Destruction_Minutes(x26))).

252[hyper,11,59,60,61,62,63,64,65,66,67]

Obligated_To_Sign_Destruction(Witness(x),Sign(Destruction_Minutes(x))).

253 [binary,252.1,68.1] \$F.

----- end of proof -----

This query reaches to a proof after we add more real world facts as clauses to the input file submitted to the OTTER. CPU time used is twice as much of the previous queries. The proof requires the witness to sign the destruction minutes of the classified document after destruction.

6. Query 6

Query 6 in Q_{NL} :

As a head of the sending office, I am required to sign the consignment note attached to the classified document. Can I delegate to the authority to sign the establishment note to another agent working in the sending office?

The contradiction is searched in this query is “The head of the sending office can not delegate her signature authority to sign consignment notes to another agent working in the sending office.” Q_{NL} is converted into Q_{COMP} , to test the query in theorem prover. The Q_{COMP} of this query is:

$\neg(\text{exists } o \text{ (exists } d \text{ ($

```

Organization(o) & Sending_Office(o) & Classified_Document(d)
&
Decide_To_Send(o, Decide( Send(d))) & Consignment_Note(d)
& Representative(d) &
Works_For(Representative(d),Head(Sending_Office(o)))
->
Permitted_To_Delegate_Sign(Head(Sending_Office(o)),
Delegate(Representative(d),Sign(Consignment_Note(d))))
))
).
```

The query is executed in OTTER, the result of the proof in a concise form is:

----- PROOF -----

```

7 [] -Organization(x7)| -Sending_Office(x7)| -Classified_Document($f4(x7))|
-Decide_To_Send(x7,Decide(Send($f4(x7))))| -Consignment_Note($f4(x7))|
-Representative($f4(x7))|
-Works_For(Representative($f4(x7)),Head(Sending_Office(x7)))|
Permitted_To_Delegate_Sign(Head(Sending_Office(x7)),
Delegate(Representative($f4(x7)),Sign(Consignment_Note($f4(x7))))).
59 [] Organization(x26).
60 [] Sending_Office(x26).
61 [] Classified_Document(x27).
62 [] Decide_To_Send(x26,Decide(Send(x27))).
63 [] Consignment_Note(x27).
64 [] Representative(x27).
```



```

65 [] Works_For(Representative(x27),Head(Sending_Office(x26))).
66 [] -Permitted_To_Delegate_Sign(Head(Sending_Office(x26)),
Delegate(Representative(x27),Sign(Consignment_Note(x27)))).
370[hyper,7,59,60,61,62,63,64,65]Permitted_To_Delegate_Sign(
Head(Sending_Office(x)),
Delegate(Representative($f4(x)),Sign(Consignment_Note($f4(x))))).
371 [binary,370.1,66.1] $F.
----- end of proof -----

```

This query reaches to a proof. One of the reasons behind the proof is the already proved query(i.e. Query₁) CPU time used is five times as much of the previous queries. The head of the sending office can give the authority to a representative. The execution of the query reminds the following:

- The unproved queries can be proved by adding additional real world facts and running simpler queries before complex ones.
- OTTER can not create the exact schema of the relationships beforehand even if a schema-based approach in structuring policy proposed in this thesis is used. Human intervention is needed .

THIS PAGE INTENTIONALLY LEFT BLANK

IV. POLICY TESTING

A. INTRODUCTION TO POLICY

Policies are statements of goals, or rules or guidance governing the actions taken to satisfy goals[See the Michael-Ong-Rowe for more on policy,36]. They provide broad direction and goals. The policy as a term is synonymous with the preferred behavior.

Linington[8] treats policy as a prescription that covers a number of actual choices. Policies can also be considered as a statement of what to do after observing a series of ad hoc decisions being taken inconsistently. Linington defines meta policy as a statement that can define when the policy is applicable, what information must be available for the policy to be applied (or the objects representing the policy enforcement), defining the decision process of what to do if a policy is violated or is inconsistent with another policy, defining the decision process of how to evaluate adherence to the policy, and defining all invariants that must be adhered to for the policy to be effective.

According to Pfleeger[22], security policy is a statement that should specify an organization's goals regarding security (e.g., protect data from leakage to outsiders, protect against loss of data due to physical disaster, protect integrity of data); where the responsibility for security lies (e.g., with a small computer security group, with each employee, with relevant managers); and the organization's commitment to security. In other words, security policy is the subset of an organization's policies that pertain to the protection of information and computing resources from unauthorized access. Security requirements for an information system are derived from security policy.

B. POLICY AS A MOVING TARGET

Policy is a moving target: the policy of organization changes over time. Thus, there is a temporal validity associated with both policy objects and the

relationships between policy objects. Policy changes can be initiated within an organization, or by parties external to the organization, such a developer of a commercial-off-the-shelf software application: such products have policy embedded in them. Thus, an organization needs a means for updating its policy base. However, to do so, an organization needs a policy workbench to help it cope with the size and complexity of its policy bases. Moreover, the policy workbench should have rollback and recover capabilities to control updates. Configuration management of the updates is also critical to ensure authorized updates. Otherwise, the effort to update the existing policy base will end up mixing up the current existing policy base.

C. MAINTENANCE OF POLICY

Before operationalizing a change to the policy base, the organization needs to assess the potential effects of those changes. In order to assess the effects of proposed updates to a policy base, either the user of policy workbench or one of the workbench-based tools needs to pose questions about the change. The queries, in turn, serve as basis for testing policy.

- The policy user or a workbench tool submits an update or query about policy.
- Queries are posed by humans converted from Q_{NL} (Query Natural Language) to Q_C (Query Computational).
- Automated test generator creates the necessary test for that query.
- One or more testing tools generate and execute test plans.
- The testing tools send their test results and assessment of the proposed change to policy to the user interface module of the policy workbench.

One of the requirements we envision for the testing tools is that they provide for the systematic and repeatable testing of the policy. After the first round

of testing, one might find gaps; this requires the correction of gaps and rechecking the revised policy base. Testing tools can reuse testing results or test cases, or generate new test patterns and cases.

D. TESTING

IEEE STD 829-1998 defines testing as the process of analyzing a software item to detect the differences between existing and required conditions and to evaluate the features of the software item[24]. According to Myers[26], testing is the process of executing a program with the intent of finding errors. In the context of policy, the goal in testing is to detect and identify gaps in the policy base.

Software testing can only be formalized and quantified when a solid basis for test generation can be defined. Tests are commonly generated from program source code, graphical models of software specifications (e.g., control flow graphs), and requirements. The testing process needs to be congruent with that of the software development process. In addition, the testing mechanism to be used for testing policy needs to describe how the functions the software provides are connected in a form that can be easily manipulated by automated means. According to Binder[25] “it is the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs.” A well-structured, systematic and repeatable approach will, in theory, result in a maintainable test suite and a maintainable policy base. Martin and McClure[25] stated that operations and maintenance represent up to sixty-seven percent of the system life-cycle. Thus, it appears that automated tool support for testing policy could be of great value to an organization.

Using an ad hoc approach will result in a failure for diagnosing gaps, for instance inconsistencies in the policy base. Kaner[27] states that testing is the second most costly activity at the initial development of a product.

1. What are the Steps When Testing Policy?

Using the test design steps, the policy can be tested with the following steps:

- Identify, model, and analyze the goals and responsibilities of the policy under test (PUT).
- Design test cases based on the external specifications of the policy.
- Add test cases based on heuristics.
- Develop expected results for each test case or choose an approach to evaluate the pass/no pass status of each test case.

2. Development of Test Cases in Policy Testing

A test case is a set of inputs, execution conditions, and expected results developed for a particular objective (Fig. 11). According to Myers[26], one of the most important considerations in testing is the design and invention of effective test cases. The subject of test-case design is considered to be the crux of software testing. The reason for this is that “complete testing” is impossible, that is testing of any program for all possible reachable states is an np-complete problem. In order to test software adequately but within a short time frame, selecting the proper subset of test cases is a necessity. Hence, the tester or computer-based testing tool must determine the subset of test cases that has the highest probability of detecting the greatest number of errors per unit time and at least cost.

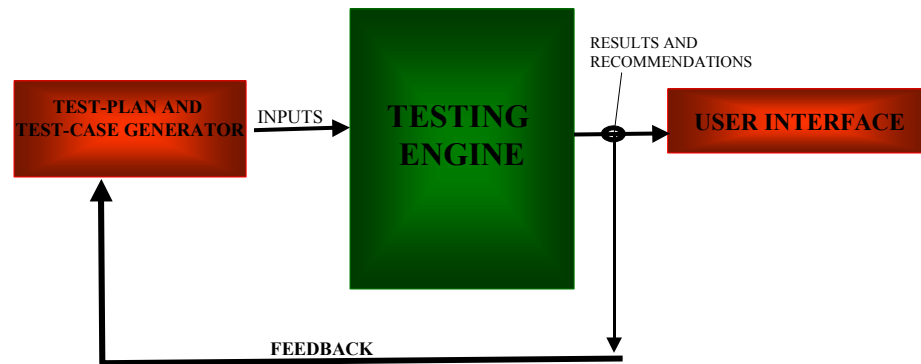


Figure 11. Test Case Specification Diagram

Test-case-design methodologies can help in determining the subset of test cases to use in testing policy. Some of the desirable properties of a test case are as follows[24]:

- Exercise as many different inputs as possible. In terms of polic, the goal is to develop test cases that exercise as many of the policy objects as possible.
- Covers a large set of other possible test cases. By partitioning the input domain into a finite number of equivalence classes, one can select test cases that are representative of that equivalence class.
- Tells more about the specific set of input values.
- Has a reasonable probability of detecting gaps.

- Is not redundant. Is neither too simple nor too complex. Starting with simple test cases will give the tester each module's gaps. The tester needs to see the outcomes when combining more than two test cases are executed. If only the simple test cases are executed, these gaps related to combinations of two or more test cases cannot be detected. If the test cases are too complex, the outcomes are insurmountable for the tester.

E. TEST PATTERNS AND REUSABILITY

Test-design patterns developed according to the above steps can be reused. Reuse is one means for addressing issues associated with system cost, maintenance, and adaptability[25]. Object-oriented technology provides explicit support for reuse: component can be designed to support a broad range of applications and be tailorable.

Policy designed and structured from an object-oriented model of the policy objects and relationships facilitates reuse. An obstacle can be the use of different types of object-oriented models for the same policy base. The Unified Modeling Language(UML) can be used to address the problem of using different models each with different model semantics. Organizations do not require structuring and designing each subbranch policies from scratch as a separate process. UML diagrams enable one to view policy at multiple levels of abstraction. The transition among UML diagrams eases the process of switching among the multiple levels of abstraction in which policy is represented. Transition among UML diagrams is discussed in [12] and [25].

Developing test patterns for the SUT can facilitate testing[26]. The key idea here is to create tests from extant patterns represented in a common language: the test team can leverage the use of test cases throughout the organization rather than having to integrate test patterns from models with disparate semantics. Moreover, once the organization creates libraries of test

patterns, the suborganizations can use those patterns directly or adapt those patterns to their specific policy base.

In this thesis, we reuse test patterns derived from UML models of policy.

F. STRATEGY FOR DEVELOPING TEST CASES

Test cases are defined with test case specifications. According to IEEE Std 829-1998, the test case specification includes the following sections:

- Test case specification identifier. A unique number or name
- Test items. The features and modules under test are identified. References to specifications and manuals can be made here.
- Input specifications. This section includes the list of all inputs by their value interval, and name if they are files. It includes the inputs passed by operating system, supporting programs or databases, prompt messages displayed, and relationships between them. Timing considerations also should be described herein[27].
- Output specification: a list all messages and outputs.
- Environmental needs. Special requirements such as hardware, software, and staff should be listed here.
- Inter-case dependencies. What are the dependencies between this test case and the others? If the ones executed before this one fails, what will be done next?

The strategy for testing is just like an umbrella over the testing process, guiding of all of the testing activities. The place of test strategy and test cases in the testing process is shown in Figure 5.

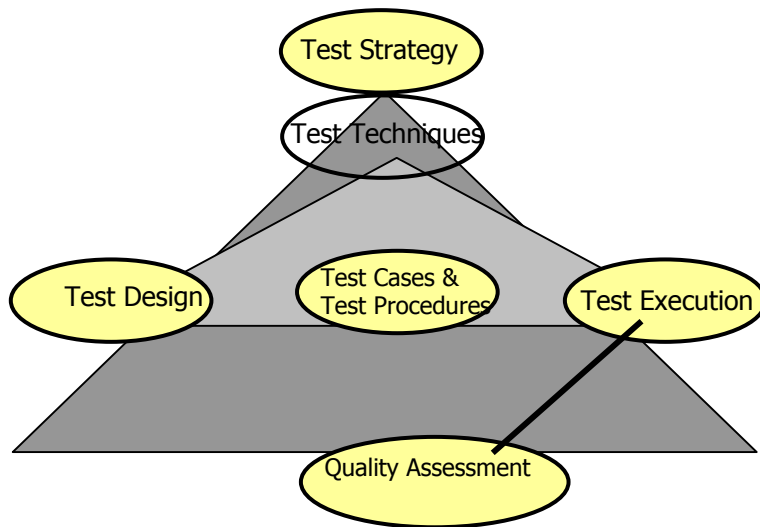


Figure 12. Test Case Specification Diagram[27].

One relatively weak strategies for creating test cases is to rely on the capture feature of an automated testing tool. The problem associated with capture is analogous to the problem posed by inserting constants into the body of programs. The capture feature takes the exact sequence of keystrokes, mouse movements, or commands and turns them into capture. These captures are just like constants. It is advisable to avoid using constants in the body of program. By the same token, one should avoid using capture tools. Most of the code that is generated by a capture utility is unmaintainable and of no long-term value. However, the capture utility can be useful when writing a test because it shows how the tool interprets a series of events.

Another weak approach involves the creating of test cases without a common goal. Such an ad hoc approach can result in the creation of repetitious in scripts, which in turn can lead to a poor use of testing resources.

In contrast, the testing strategies that work well can be divided into two paradigms data-driven and frame-based architecture[27]:

In the data-driven testing paradigm, the usage of test matrices is a key aspect of the data-driven testing paradigm. Each row in a test matrix specifies a test case and each column represents one of the parameters. The execution of the

test cases can be as follows. A script reads all of the parameters required for the execution of that particular test. Once all of the parameters have been set, the script executes that command. As the code is separated from data, the developer is not required to modify each test case. The subject matter of data-driven automation strategy can include the following [33]:

- Parameters for inputting to the program;
- Sequences of operations or commands that make the program execute;
- Sequences of test cases that drive the program;
- Sequences of machine states that drive the program;
- Documents the program can read and operate on;
- Parameters or events that are specified by a model of the system (such as a state model or a cause-effect-graph based model).

The developer can select one of the interfaces to enter data into a file and provide different interfaces for testers with different needs and skill sets [33]. Data-driven approaches are highly maintainable and easy to work with; test matrices assist the tester in assigning tests within many platforms.

There are multiple methods used for data-driven approaches. [34] uses a spreadsheet of commands. One row contains sequence of commands required to execute a test. The benefit of this technique is as the user interface changes, instead of rewriting the test code, one just needs to modify the spreadsheet. Some other approaches use simple test cases of machine states.

The framework in a framework-based architecture paradigm consists of a set of functions in a shared library and located between the test scripts and the SUT. Thus, the scripts can be programmed independently, regardless of the user interface of the software. The scriptwriters can use that set of functions as if they are commands of a programming language.

Choosing small, conceptually unified tasks as functions helps a lot during scripting and maintaining scripts. For instance, the `openfile()` function is an example of a frequently used function in the library. Having the `openfile()` function in the library will not only save the scriptwriter time but also improve the maintainability of the scripts.

In addition to the functions, Arnold [35] proposes that scripts be used in conjunction with wrappers(i.e., a routine that is encloses another function). By adding a logging function or an error handler or a call to a memory monitor, for example, the script gains new capabilities without the need for additional code. Wrappers can be very useful in stress testing, test-execution analysis, and reporting and debugging.

One weakness of the framework approach is that building all of the commands into the library requires a large staff of testers. Kaner notes that allocating the testing staff to create the programming library may cause a failure in the automation of projects[27]. Another point is that expecting scriptwriters to use the library just because it is there is not realistic. Some programmers are not inclined to use the code that they did not write. In short, there is a need to manage the library.

In summary, the framework approach is more tedious than the data-driven approach to testing.

G. PROPOSED APPROACH TO POLICY TESTING

The task of using theorem provers, expert systems or any other tools to test policy involves a lot of formula manipulations and computations that are best tied to mechanical means. One of the viewpoints in the testing is since it is impossible to prove the absence of gaps, then demonstrate that the policy is free of gaps for specific test cases. The difficulty in this thinking is the enormous intellectual effort that is required. For example, in a large organization like the U.S. DoD., its suborganizations are distributed throughout the U.S. and the world.

Each branch (Army, Air Force, Navy, Marine Corps, and Coast Guard) has its own set of policies in addition to the global DoD policies. Sub branches also have their own policies in compliance with the global policy. One of the sub branches, such as the Naval Postgraduate School, can add a policy. Before the school implements the policy, it needs to test whether the policy:

- is consistent with extant policy
- is sound in the sense that the policy is not subsumed by extant policy
- is complete, in the sense that there is at least one policy object that is related to a policy object in extant policy base.
- creates any other types of gaps in the policy base

1. Unautomated Testing Process

If there is not any automated testing in place in the organization, the process can be as follows:

- Add policy is described in natural language
- The person in charge converts a proposed policy from natural language to computational language
- Adapt the computational form at hand to each separate testing tool and invoke them in the tool-context

This part is the most tedious and cumbersome. Because the person in charge of adding policy must know how to invoke all of the tools at hand and have the ability to develop a computational form specific to these tools. Thinking that every staff in charge of policy add/delete/update operations is at the same level of expertise throughout an organization is naive.

- Retrieve the results from the test tools
- Interpret the results

As different tools are used and each tool has its own way of handling tests, the tester must interpret the outcomes of these tools separately, because each tool produces outputs in its own format. In order to evaluate results from different tools, the staff in charge must be able to understand the results and then express them in a common format.

- Convert the test results into a natural language
- Compare results.

As the above-mentioned process heavily depends on the abilities of the staff in charge of testing, the process of updating policy can result in a highly variable situation. Subjectivity and limits of the staff becomes a problem. The alternative is to automate the process, replacing the staff-dependent process by a systematic and repeatable automated process.

2. Automated Policy Testing Process (Our Testing Strategy)

The following discussion of our strategy refers back to Figure 13:

- The end user who is responsible for maintaining(i.e., updating) submits, in a natural language, a policy and intended update operation(e.g., add, delete, change, revert) or a query about the extant policy.
- The update request or query is converted by a natural language processing tool into a computational form.
- An automated test generator creates the necessary test for that update operation or query.
- Test is conducted.
- The results of test and any associated recommended corrective actions are presented to the end user via natural-language processing interface.

The end user only fills out the query and gets the results. The commands and their syntax for invoking testing tools are transparent to the user of the policy workbench. Moreover, the user is not required to prepare the query and express it in computational form later. Thus, the user is relieved of the mechanical tasks of translating and testing policy and can concentrate on performing higher-level tasks, such as making policy, or interpreting policy, or decisions based on policy. The automated test generator picks up the outputs of the tools and presents it in two different interfaces. In one of the windows, the summary of the output from the tool and recommended action is presented to the end user. In the second window, the detailed output from the tool is presented to the end user.

Our testing strategy is to make the set of support domain used by a theorem prover to be as small as possible. The theorem prover is used to test for logical inconsistencies between policies. For discussion of set of support see Appendix A. Even a small program can have 10^{14} unique paths[26]. Policies are much more complex than the small program. Trying to search for all the branches in a policy base is an intractable problem unless the reasoning program focuses on keys that help to narrow down the scope of the search. Moreover, an exhaustive testing strategy does not equate to complete testing of all of the relationships among policy objects. The reason is that an exhaustive search does not guarantee that the policy base is free of logical contradictions between policies. Another reason is there might be missing relationships between policy objects that precludes the discovery of a logical contradiction. Hence, testing an incomplete domain will not result in complete results. On the contrary, narrowing the paths enables the program to concentrate on the domain of interest to the user of the policy workbench.

We also propose to generate tests automatically from the software design (i.e., in this thesis policy design), rather than the code. Policy designs provide a basis for testing how the software functions behave in a form that can be easily manipulated by automated means[31]. Generating test data from high-level policy-design notations structured in UML has several advantages. First of all, design notations can be used as a basis for output checking, which can reduce one

of the major costs of testing. Secondly, the process of generating tests from design can point to gaps in the design itself. Thirdly, generating test cases during the desing of policy allows for testing activities to be shifted to earlier stages of development. Lastly, the test data is independent of any particular implementation of design.

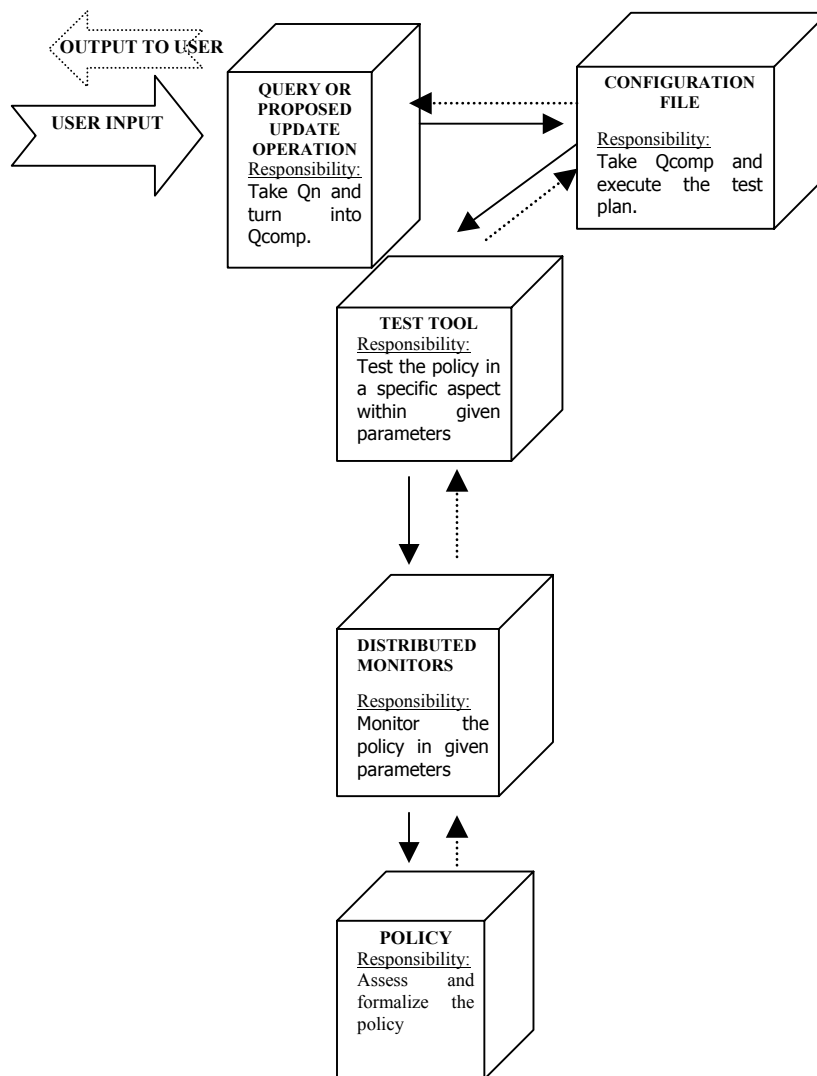


Figure 13. Engineering Diagram of Our Automated Testing Process.

3. Benefits of Our Testing Approach

The benefits of the automated test generator that is developed in this thesis are as follows:

- Systematic way of maintaining policy.
- Unified approach for applying policy throughout an organization.
- Internal details of the testing are hidden from the end user.
- The process of invoking tools is hidden from the end user.
- Turns exhaustive search into a directed search strategy.
- Increase the probability of finding a relevant answer to search as the search is executed in the domain under focus.
- Assists the end user in achieving a high-level of productivity by automating tedious mechanizable tasks associated with testing policy.

V. RELATED WORK

A. DISCUSSION

1. Policy Specification and Axiomization

Cuppens and Saurel[2] investigate how to specify a security policy. Cuppens and Saurel explore how to formalize security policies in such a way that administrators can automatically derive the consequences of the policies in place. Cuppens and Saurel have chosen a logic-based approach in an attempt to keep their approach as domain-independent as possible.

In particular, the authors use deontic logic to formalize a set of corporate security policies via a case study. Deontic logic can be used to specify the concepts of obligation, permission, and prohibition. The other reasons for selecting a deontic-logic-based approach are as follows:

- Deontic logic is a universal formal language, useful for expressing any kind of knowledge or data
- Tool support exists for deontic logic which support a logical theory (several kinds of PROLOG and other theorem provers)

There is a clear connection between our work and the approach proposed by Cuppens and Claurel. For above-mentioned reasons, deontic-logic-approach is used in this thesis. However, in this thesis we do not try to tailor our testing strategy or approach to a particular type of policy.

Cuppens and Saurel also shed light on expressing a natural-language statement of policy in a formal way and the difficulties encountered during their research. One of the main difficulties is the natural complexity and ambiguity of a text in a natural language. Obtaining a correct and as precise as possible interpretation of the policy at hand becomes crucial in this respect. An organizations may use somewhat imprecise wording in their policies so that the interpretations of the policies can vary from one suborganization to another.

However, the authors claim that it is difficult to automate the process of giving domain interpretations of the policy.

In addition to Cuppens and Saurel, Moffet and Sloman[3] also proposed deontic logic as a sound approach to formalize policy.

Michael, Sibley and et.al.[1] investigate the formalization of security policies using a logic-programming paradigm. The authors assert that one should use a structural model of policy objects and relationships to guide the axiomization of policy. In their case study, they compare model-based approach with one with no pre-structuring (unstructured approach). Both of the approaches used predicate calculus. In the structured approach, an Extended Entity Relationship Diagram is used to develop the schema. In the ad hoc approach, the security policies are axiomatized without the aid of schema.

The results of the schema-based approach provides a common referent for terms used in rules and the relationships between the rules. For the unstructured approach, the number of axiomization errors increases as the axiom base is refined.

We adopt UML notation is used to model the objects and the relationships in the policy.

2. Policy Testing

Cuppens, Saurel and Carrere[4] propose an approach to detect and resolve conflicts between normative policies. They define the security policy in a broad sense; as of set of norms to be enforced by agents. The authors address the following topics:

- Semantic heterogeneity
- Domain constraints
- Completeness problem
- Conflicting norms problem

Cuppens et.al suggest a language to specify regulations and show how to use this language in a case study. In our thesis, we follow some strategies proposed by authors for axiomizing policy in deontic logic. In particular, the authors provide strategies for resolving normative conflicts when merging different policies. Firstly, they created a hierarchy of strategies to resolve conflicts: atomic strategy, meta strategy, and good strategy. Then, they applied these strategies to resolve conflicts to obtain a global non-conflicting security policy base.

Schmid, Ghosh, and Hill[5]’s main objective is to develop test generator to test the robustness of Windows NT . They develop both a binary wrapping and a fault injection technique to test for robustness. The authors use a configuration file that triggers the automation of the robustness testing of Windows NT. In this thesis, the configuration file takes the axiomatized policy base , and based on queries invokes the the related tools with the required parameters.

Schmid, Ghosh, and Hill assert that the wrapper serves as a tool that simulates the effect of failing system resources, such as memory allocation, network failures and file input/output problems. We use their general philosophy to “wrap” policy, much as the authors use software wrapper to simulate the conditions under which the policy fails.

Kim and Carlson[6] investigate the generation of test plans during the design stage of system development. They propose three software testing metrics for object-oriented Software Development:the most critical scenario, the most reusable component, and the most reusable subpath.

Kim and Carlson used interaction diagrams to identify the relationships among the objects in the testbench. They derive use-case matrices from the interaction diagrams, a collection of specific variants of an executable sequence of use-case actions.

In this thesis, we use interaction diagrams to direct the application of testing to specific subsets of the policy base.

3. UML as Policy Representation Framework

Putnam[7] introduces a representation of policy based on the ISO/ITU Reference Model for Open Distributed Processing (RM-ODP). They investigate how certain behavior can be specified and achieved in an object-based system, using the constructs of the Unified Modeling Language (UML) and Object Constraint Language(OCL).

The focus of Putnam's work is on policy for fault tolerance. Specifications of contracts and policies provide a baseline against which to describe deviations from expected behavior of a system. A fault-tolerance policy is defined as a rule about a failure mode that is not allowed to occur.

Putnam represents fault-tolerance policy in UML in conjunction with object and processing aspects of the fault-tolerance mechanisms, to achieve a good specification that includes some of the behavioral semantics of fault tolerance. Putnam describes some limitations with using UML to represent the full semantics associated with fault-tolerance policy. Putnam also discusses the lack of tool support for represent policies with full semantics.

Linington[8] explores the requirements for defining communities and expressing policies within a UML environment. The author mentions the weakness of UML in supporting the combination of existing, parameterized specifications and, in particular, for defining and managing policies.

The author puts forth the idea of invariant in the policies to keep the policies as short and manageable as possible. Linington states that the statement of objective in ODP can not be satisfied within UML: additional forms of a constraint language are required. However, a sound strategy for expressing policy in UML is given by Linington. In the representation of the policy using UML, the options for representation are as follows:

- Defining Policies as Classes

The author of a policy defines a new class, which is slotted into the specification and defines the detailed behavior of the policy. The challenge is to

try to construct a sufficiently narrow interface to make substitution safe without loss of expressive power.

- Defining Policies as Communities

This approach follows the same style as above, but adds the discipline of the ODP community and role framework to declare and explicitly structure the elements affected by the policy.

- Defining Policies as Diagrams

Another approach is to partition each policy from the remainder of the specification by making use of the presentation structure. As compared to the others, this approach puts little restriction on the policy specification.

Linington compares and contrasts the foregoing alternatives.

Voas and Miller[9] introduce the PIE (Propagation analysis, Infection analysis and Execution analysis) technique, which can be used for predicting the fault-revealing power of test cases. Data about test cases are used to create histograms. Each bin in a histogram represents a single test case. The score in a bin predicts the likelihood that the test case will reveal a fault through the production of a failure (if a fault exists in the set of program locations that the test case executes).

The authors' preliminary experiments suggest that the histogram technique that they present in this paper can rank test cases according to their fault-tolerant revealing ability. The application of the technique results in the grouping of test cases into a suite according to each test case's revealing capability.

Evans[17] proposes a rigorous analysis technique for UML based on the use of diagrammatical transformations. Evans uses the class diagrams to perform a number of deductive transformations. By leveraging the intuitive and understandable nature of UML and object-oriented methods, the author used these diagrams for developing deductive transformation rules. Although Evans' work is

limited to class diagrams, it may be possible to reason with other UML diagrams in a similar fashion, thus making it possible for practitioners to use UML without relying on complex linguistic techniques.

In this thesis, we apply both deontic language and UML notation. UML diagrams provide patterns from which to identify the relationships among policy objects.

Selonen, Koskimes and Sakkinen[19] discuss various general approaches and viewpoints of model transformations in UML. Model transformations are essential for model checking, merging, slicing and synthesis. As UML provides various diagrams at different perspectives or abstraction levels, transformations among UML diagrams can automate a substantial amount of model transformation operations in both forward engineering and reverse engineering.

The authors' idea of transformation among UML diagrams eases the process of merging different policy objects. By using the authors' transformation approach, the conflicts and violations caused by policy operations can be minimized. In this thesis, transformations of UML diagrams help in the automation of different query types to different types of policy objects. Considering that large organizations can have a large number of sub-organizations and branches underneath, the transformation will enable small branches to increase or decrease their abstraction levels in UML diagrams. Branches need detailed UML diagrams to specify the actual setting of their policy in detail but they have to update their notation and diagrams for organization-level policy.

Favre and Clerici[18] defines a forward engineering method that facilitates object-oriented code generation from UML models. Favre and Clerici try to embed code generation within a rigorous process that facilitates reuse. Their code generation process consists of three main phases:

- The construction of an incomplete algebraic specification from UML models.
- The construction of a complete algebraic specification.

- The transformation of the specification obtained into object-oriented code.

The work of Favre and Clerici differs from that described in this thesis in that they use a semi-automatic process from UML class diagrams to algebraic specifications in the GSBL language; we use UML diagrams to structure policy and describe the architecture.

Shroff and France[13], tries to formalize the primary UML constructs used to build class structures. The authors use Z notation to express the meaning of UML class structures. In other words, their intent in integrating Z and UML is to create machine analyzable UML models through the generation of Z specifications. They propose a formal semantic base for UML class models. Shroff and France aim to fill the lack of firm semantic foundations for object-oriented modeling.

The paper does not relate to this thesis directly. But it helps in understanding the bridge between the UML diagrams and formal semantic base. The authors covered the associations, aggregations and generalization in a clean way. Their work paves the way to a full semantic base for object-oriented design.

Selic describes the usage of UML in real-time systems to predict system properties and consider physical logical resources before fully implementing a system. This paper does not relate to this thesis directly, but could be used to consider issues related to expressing and testing policy regarding real-time systems.

4. Formal Methods in Policy Structuring and Testing

Linington, Milosevic, and Raymond[28] extend the Reference Model for Open Distributed Processing (RM-ODP), and introduce the notion of an enterprise viewpoint, and provide a minimal set of concepts for structuring enterprise-language specifics. The authors explore how policy can be modeled between and within communities. In particular, they investigate how policies in different communities interact and how a community can impose policies.

The deontic logic is the way that we express policies in this thesis. The authors defined deontic logic, the challenges with using deontic logic. Permission, obligation, and prohibition are defined examples are given of these terms-encompass. They give real-world examples of permission, prohibition and obligation. Differences among these terms is explained. Policy making, creating policy frameworks, and enforcing policy are the other notions explained in the paper.

The authors' work sheds light on this thesis by expressing the permission, prohibition, and obligation in a concise manner. Their work differs from this thesis in such a way that their research is focused on ODP and policy making in ODP. The authors argue for the use of communities to aid the process of policy representation. One way of expressing policy is by expressing policy objects as communities. In contrast to the foregoing approach, in this thesis policy representation is performed via class diagrams and sequence diagrams.

Kim and Carrington[29] explore the visualization formal specifications. They mentioned that formal notations provided by most formal specification techniques are not easy to use and understand by the general populace. Kim and Carrington try to visualize both the static and dynamic aspects of the formal specifications.

The authors use Z specifications as an example and visualize these specifications in UML. Even if they suggest the use of Object Constraint Language (OCL) to supplement the lack of expressiveness of the graphical notation to specify complex constraints precisely, they prefer to express these constraints diagrammatically.

During translation from Z specification to UML class diagram, they analyze the semantic relations between variables declared in the state schema and the UML class constructs. Later on, the variables are translated to the most appropriate UML class constructs depending on their semantics. The diagrams the authors produce are just as complex as the formal specifications. The

expressiveness power of the diagrams is weak, Kim and Carrington lay the groundwork for visual representation of formal specifications.

The focus of this thesis differs from that of Kim and Carrington in that UML is used to structure the policy and define the relationships among different policy objects to guide the formalization policy, rather than attempt to visualize an existing formalized policy into a visual representation.

Ober[18] proposes an executable profile for UML, that will add rigor and allow some advanced features to be added in supporting tools. Ober choose Abstract State Machines (ASM), or also known as evolving algebra, to create the executable profile. The advantage of the ASM is the ability to describe the semantics of UML. Ober lays the groundwork for executable profiles.

The author's work relates to this thesis in the following ways:

- Their effort concentrates on increasing the expressive power of UML. Our work in this thesis is confined to structuring and defining policy in UML.
- UML is used as a graphical notational tool in this thesis; main goal of this thesis is structuring and testing policy in an automated way rather than adding precise semantics to the language.

Ober addresses the question of “why should one build precise formal semantics into UML?”

Abdurazik and Offutt[31] introduce test criteria based on UML collaboration diagrams. Tests are commonly generated from program source code, graphical models of software (such as control graphs), and specification/requirements. The authors propose using collaboration diagrams to represent a significant opportunity for testing because they precisely describe how the functions the software provides are connected in a form that can easily be manipulated by automated means. They use collaboration diagrams to describe

how the objects communicate. In particular, they create a model for performing static analysis and generating test inputs from UML collaboration specifications.

The authors assert that tests can be generated automatically from the software design rather than the code or specifications. So authors introduced new integration-level analysis and testing techniques that are based on design descriptions of software component interactions.

In this thesis, we also use collaboration diagrams for testing policy.

VI. CONCLUSIONS AND FUTURE WORK

A. SUMMARY

Organizations are policy-driven entities. Policy bases can be very large and complex. Thus, maintenance of policy and the assurance of the consistency, completeness, correctness, and soundness of a policy base necessarily requires some level of computer-based support. Policies of an organization change: there is a temporal validity associated with policy. Thus, it is necessary to test a policy base each time it is updated.

A policy workbench is an integrated set of computer-based tools for developing, reasoning about, and maintaining policy. A workbench takes as input a computationally equivalent form of natural-language policy statements. Each of the component tools can manipulate the computational representation of policy.

We developed an object-oriented schema-based approach to structure operational policies. Our structural model consists of Unified Modeling Language (UML) class and collaboration diagrams.

A complete structure that has the full semantics of the natural-language policies cannot be achieved with the full semantics within the UML framework. To fulfill this inadequacy, a precise semantic description is required. We chose to use deontic logic in order to model the semantics of permission, prohibition, and obligation, as they are used in policy. Policy includes the following that carries the specifications of a normative system:

- When the policy is applicable (i.e., time)
- What (e.g., information, resources.) allocated to carry out policy
- How is the degree of adherence to policy to be evaluated
- What should be done in the event of violation

In addition, deontic logic is a universal formal language, for which automated reasoning systems can take deontic specifications as input.

Starting with our structural model of policy, we develop test cases to test policy bases. The proposed update to the policy base or query about policy dictates the content of its corresponding test cases. The Policy Base Under Test(PBUT) is analyzed to determine the test cases that need to be generated and run. The PBUT is searched by temporal property, counting property, sequence property, and the combinations of these properties. The combinations of these properties are used as test patterns for directing testing.

A practical solution to test-case development is provided. The test spectrum has query-specific tests at one end, and the generic types of tests at the other end. Different kinds of queries are created to test the policy base across the spectrum under a range of scenarios. The policy base is dynamic, so the tactical testing needs to be changeable on-the-fly. We introduce pattern recognition in the test-case database so that a systematic and repeatable rule-based strategy can be applied. Heuristic rules guide the automatic test tool. We give examples of heuristic rules that can be used in test-case selection and generation.

We investigate the use of statistical inference of reuse of test cases by determining the patterns that approximate the query-to-be-executed. The patterns that are selected are then used to determine which of the tests in the information repository need to be retrieved and executed.

Query mapping among queries (intra relationship), query mapping inside a particular query (i.e., intra-query relationship) and mapping among predicates (i.e., semantic mapping) concepts are introduced in this thesis. Making these mappings explicit can increase the likelihood of identifying gaps during the testing of policy.

Our testing strategy provides a starting point for a automatically testing of operational policy. And confirms evidence for automatically generating test cases for operational policy by utilizing test patterns. However, there is still a need for human intervention to reason about the effectiveness of the testing and take

corrective actions. For instance, the set of support provided to a theorem may need to be narrowed or broadened to test for logical consistency within a policy base. In other words, we propose a systematic way of creating a good first approximation of the necessary test cases by utilizing test patterns. However, the sufficiency of these test cases relies on automated processing of heuristics and human judgement.

B. FUTURE WORK

Future research topics include the following:

- Try to implement an object-oriented schema-based approach to structure much more complex policy bases.

In this thesis, the object-oriented schema-based policy approach is introduced and used in a compact policy base. The approach is used to structure one policy base. The sample implementation in this thesis paves the way for the structuring of multiple policy bases. An area of future research would be to investigate how to express inter- and intra-policy base relationships and the collaboration diagrams that shows the details of these relationships.

- OTTER is used as a tool in this thesis to determine conflicts, inconsistencies, and gaps. Try to use other first-order logic tools to test the PBUT.

One of the reasons behind the selection of first order logic in this thesis is that it is a universal formal language for which automated reasoning systems can take deontic specifications as input. The goal of a future investigation could be to take the policy base in this thesis and implement it in different first order logic tools(e.g., METEOR from Duke University, SETHEO, LeanTaP, Bliksem), then apply testing tools or strategies that best lend themselves to these formal models.

- Try to implement the statistical inference that is applied for the case study for a complex policy base with a large information repository of possibly diverse test cases.

A statistical analysis of the history of processing test patterns could be used to characterize the PBUT. In this thesis, the concept is explained and a small example is given. A real-world example remains to be generated.

- Try to implement fuzzy logic in a multiple policy base.

Fuzzy logic brings a systematic way to reason under uncertainty. Structuring multiple PBUTs involves fuzzy reasoning by its very nature. Future research could take the form of representing multiple PBUTs in fuzzy logic and reasoning about them. Policy-making is a human enterprise, integrating many complementary, contradictory, fuzzy, and changing human values. It may be the case that the use of fuzzy logic is necessary for representing and reasoning about certain types or instances of policy.

- Try to implement the methodology used in this thesis to goal-type policy.

Because goal-oriented policies differs from operational policy, one would need to ask the following type of questions:

- How can we test goal-oriented policy?
- What would be the purpose of testing for goal-oriented policy?

APPENDIX A OTTER TUTORIAL

OTTER(Organized Techniques for Theorem-proving and Effective Research) [28] is the theorem prover that is used in this thesis as a testing tool. This appendix provides an overview of OTTER, with particular attention paid to those aspects of OTTER that are integral to our approach to testing for gaps in policy. For an introduction to automated reasoning see[29]. For a tutorial on how to use OTTER see [30].

OTTER is a first-order logic with equality theorem prover. It is coded in C, and its source code is publicly available from Argonne National Laboratory. OTTER recognizes two basic types of statements: clauses and formulas. OTTER's search for proofs operate on clauses. Formulas are first-order statements without free variables (i.e., all variables are explicitly quantified).

A. HIGHLIGHTS OF OTTER

- % starts a comment. Comments are not echoed to the output file.
- . (period) terminates input expressions.
- () [] { } and , are grouping symbols. OTTER does not treat a comma as an operator.
- Unlike Prolog, OTTER requires whitespace in some cases. The rule of whitespace is "Insert some whitespace if and only if it is not a standard application." In other words, two pices of whitespace in $(a + (b + c)) = (d + e)$ are required, and no whitespace is allowed after f or g in $f(x, g(x))$.
- OTTER sends most of its output to "standard output" which is usually redirected by the user to a file; can be called output file. The first part of the output file is an echo of most of the input and some additional information, including indentification numbers for clauses and description of some input processing.

The second part of the output file reflects research. The output can be controlled by various flags.

- Input to OTTER consists of a small set of commands, some of which indicate that a list of objects (clauses, formulas, or weight templates) follows the command. All lists of objects are terminated with **end_of_list**. For instance, **make_evaluable(sym, eval-sym)** makes a symbol evaluable.
- Given an expression how it looks like might be associated in a number of ways, because the relative precedence of the operators determines, how it is associated. A functor with higher precedence is more dominant (closer to the root of the term). For instance, the functors \rightarrow , $|$, $\&$, and $-$ have decreasing precedence; therefore the expression **p & - q | r \rightarrow s** is understood as **((p & (-q)) | r) \rightarrow s**.
- OTTER accepts clauses both written pure prefix form and infix (abbreviated) form. For instance, Pure prefix: **| (-a), |(=(b1, b2), -(=(c1,c2))))**
- Infix(abbreviated): **-a | b1 =b2 | c1 !=c2**

Logical Expressions	Logic Symbols
Negation	-
Disjunction	
Conjunction	&
Implication	->
Equivalence	\leftrightarrow
Existential Quantification	exists
Universal Quantification	all

Table 2. Representation of symbols and operators in OTTER.

Formulas in Standard Usage	Formulas in OTTER
$\forall x P(x)$	all x P(x)
$\forall xy \exists z (P(x, y, z) \vee Q(x,z))$	all x y exists z (P(x,y,z) Q(x,z))
$\forall x(P(x) \wedge Q(x) \wedge R(x) \rightarrow S(x))$	all x (P(x) & Q(x) & R(x) -> S(x))

Table 3. Representation of Formulas in OTTER.

B. SET OF SUPPORT STRATEGY (SOS)

The set of support strategy keys on the denial clause or clauses. It seeks a contradiction that gives the theorem-proving program a convenient

termination[29]. In an automated reasoning program, the program is expected to end after it has generated a proof. To achieve that goal, the program searches for the denial of the statement to be proved and its combinations. After the search, the program designates one or more as keys. The SOS causes the reasoning program to classify newly generated clauses. The SOS strategy forbids a reasoning program from ever applying an inference rule to a set of clauses that fails to have one the key clauses present. Was defined the set of support theorem as follows:

If S is an unsatisfiable set of clause, and if T is a subset of S such that S-T is satisfiable, then the imposition of the set of support strategy on the combination of binary resolution and factoring preserves the property of refutation completeness for that combination[29].

In short, the theorem says that, if T is appropriately chosen, then the important logical property of refutation completeness of the inference rules being employed is not lost. The proof of the set of support theorem is out of scope of this thesis; the proof is given in [29].

The main advantage of applying the SOS strategy is that it makes reasoning program intelligent. Myers [26] mentioned that even a small program can have 10 trillion independent logic paths to be searched. The computation is as follows:

$5^{20} + 5^{19} + 5^{18} + \dots = 10^{14}$ where 5 is the number of paths through the loop body.

The program has a do-while loop and set of if-then statements inside it. Consider it this way: if one reasoning program could search the branches even in such a trivial program, it would take a long time and fail to find a terminating point. Real-world programs are much more complex than the simple program mentioned above. Moreover, every branch is not independent from every other branch in the actual scenarios. In other words, the number of execution paths would be somewhat fewer in real-world programs if the reasoning program has the intelligence. The set of support strategy makes the reasoning program

intelligent. As a result of that, the reasoning program is forced to concentrate on the problem domain rather than performing an exhaustive search on the general.

As an alternative method to SOS, weighting can be used to make reasoning programs operate in an efficient-manner. The problem with using weighting is that some means of assigning weights must be devised. Typically, this is left to the experience and the intuition of the user. The lighter the weight the sooner the program will look at that statement.

C. HOW DOES THE SET OF SUPPORT STRATEGY WORK?

When SOS strategy is in use, only the clauses in the SOS are chosen. According to the weighting among them, an increasing order is created inside these selected clauses. The rest of the clauses are used to complete the application of various inference rules. Then the programs continue to choose among the new clauses, those clauses that have been generated and kept because they have acceptable information. Weighting enables the program to choose among those new clauses. Therefore, the reasoning program can be forced to concentrate on specific branches.

THIS PAGE IS INTENTIONALLY LEFT BLANK

APPENDIX B WORKFLOW MODELING WITH NORM ANALYSIS

There are also other ways of structuring and defining rules in a notational language other than the methodology used in this thesis. One of them is the workflow modeling with Norm Analysis. In [37], the authors developed a modeling approach for handling business rules and exceptions to support the development of information systems. The workflow modeling is one of the methods of object-oriented information engineering, whereas Norm Analysis is a semantic approach for developing information systems[38].

The strengths the modeling approach presented in [38] are using object-oriented information engineering approach, which is a rigorous approach that provides a set of techniques for process modeling and using Norm Analysis which enables one to specify business rules that are necessary in systems design.

The first rule of the case study is:

“The sender of a classified document is obliged to update the document classification as soon as it is possible, i.e. immediately after the sender evaluates that the document classification is obsolete.”

```
(Classified_Document(d) & ( Transmitter(d) = t) &  
Res_Exec(t, Evaluate_Classification(d), level) & Classification(d) != level  
)  
->  
Obliged_to_Exec(t, Change_Classification(d)).
```

Using workflow modeling with Norm Analysis, rule 1 can be represented as follows:

Norm N1.1:

Whenever the sender evaluates that the document classification is obsolete

Then the sender of the document

Is obliged to

update the document classification as soon as it is possible

APPENDIX C SOURCE CODE OF THE IMPLEMENTATION

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% File           : Policy.in
% Author         : Lieutenant Mehmet Sezgin(Turkish Army)
% Theorem Prover: Otter 3.0.4
% Command       : Otter <Policy.in> Policy.out
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
set(ur_res).
set(hyper_res).
set(para_into).
set(para_from).
assign(max_mem,80000).
assign(max_seconds, 100000).
assign(max_proofs, 2).
set(free_all_mem).
formula_list(sos).
%formula_list(usable).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Axioms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rule 1
% The sender of a classified document is obliged to update the document
% classification as soon as it is possible,
% i.e. immediately after the sender evaluates that the document
% classification is obsolete.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
(all d
  (Classified_Document(d) &
    Transmitter(d) & Transmitter(d) = Agent(a) & Res_Exec(Transmitter(d),
      Evaluate_Classification(d), level) & Classification(d) != level
    ->
```

```

        Obligated_To_Update_Classification(Transmitter(d),
        Change_Classification(d)))
    ).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rule 2
% Any agent who is not the sender of a classified document
% is prohibited to change the classification of this document.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
(all d
  (Classified_Document(d) & Transmitter(d) != Agent(a)
    ->
      Forbidden_To_Change_Classification(Transmitter(d),
      Change_Classification(d))
    )
  ).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rule 3
% The holder of a classified document
% is permitted to ask the sender to revise the classification of this document.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
(all d
  (Classified_Document(d) & Transmitter(d) & Holder(d)
    ->
      Permitted_To_Ask_Revision(Holder(d),Ask(Transmitter(d),
      Change_Classification(d)))
    )
  ).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rule 4
% Every organization, which holds some secret documents,
% is obliged to designate an agent who is responsible for preserving these

```

% documents.

%%

(all o (exists d (

Organization(o) & Classified_Document(d) & (Classification(d) = Secret)
& Works_For(Holder(d),Employees(o))

->

Obliged_To_Designate_Agent(
Head(o),
Designate_Responsible(Document_Preservation(o)))

))

).

%%

% Rule 5

% The sender of the classified document

% is obliged to establish a entrust note of this document.

% The head of the sending side is obliged to sign this entrust note.

% The head of the sender side

% is permitted to delegate the obligation to sign the note to one of his

% representatives.

%%

(all o (exists d (

Organization(o) & Sending_Office(o) & Classified_Document(d)

->

Obliged_To_Establish_Note(Sending_Office(o),
Establish_Consignment_Note(d))

))

).

(all o (exists d (

Organization(o) & Sending_Office(o) & Classified_Document(d) &
Decide_To_Send(o, Decide(Send(d))) & Consignment_Note(d)

->

Obliged_To_Sign_Note(Head(Sending_Office(o)),
Sign(Consignment_Note(d)))

```

    ))
).
(all o ( exists d (
    Organization(o) & Sending_Office(o) & Classified_Document(d) &
    Decide_To_Send(o, Decide( Send(d))) & Consignment_Note(d) &
    Representative(d) &
    Works_For(Representative(d),Head(Sending_Office(o)))
    ->
    Permitted_To_Delegate_Sign(Head(Sending_Office(o)),
    Delegate(Representative(d),Sign(Consignment_Note(d))))
    ))
).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rule 6

```

```

% Anybody body who wants to visit a restricted area is obliged to get an
% authorized from the head of that area.
% In addition to that, the visitor is obliged to be supervised by
% an agent who is specially designated for supervising visitors.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

(
    Organization(o) & Protected_Area(a) & (Head(a)) & Person(p) &
    ( -(Work_For(Person(p),Organization(o))) &
    -Authorized_To(Head(a),
        Authorize(Person(p), Visit(Protected_Area(a))
        )))
    ->
    Forbidden_To_Visit( Person(p), Visit(Protected_Area(a)))
).
(

```

```

    Organization(o) & Protected_Area(a) & Head(a) = h & Person(p) &
    -(Work_For(p,o)) & Exec(h, Authorize(p, Visit(a))) &
    Exec_Designate_Agent(h, Designate_Responsible( Supervision(p)))

```

```

->
    Permitted_To_Visit( p, Visit(a))
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rule 7
% The holder of the secret document and the witness of the destruction
% are obliged to sign destruction time at every secret level classified document
% destruction.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
(all d
    (Classified_Document(d) & (Classification(d) = Secret) & Holder(d) &
      DESTROY_DOCUMENT(d) & Actor(d) & Destroyed_Object(d) &
      Witness(d) & After(Destroyed_Object(d)) & Destruction_Minutes(d)
    ->
        Obligated_To_Sign_Destruction(Holder(d),
          Sign(Destruction_Minutes(d)))
        & Obligated_To_Sign_Destruction(Witness(d),
          Sign(Destruction_Minutes(d)))
    )
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rule 8
% After each meeting, the organizer of the meeting
% is obliged to burn all the preparatory documents of this meeting completely.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
(all d
    (MEETING(d) & (Organizer(MEETING(d))) & After(MEETING(d)) &
      Element_Of_Meeting(d, Preparatory_Documents(MEETING(d)))
    ->
        Obligated_To_Burn(Organizer(MEETING(d)), Incinerate(d))
    )
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

% Rule 9

% The organizer of the meeting is obliged to keep these preparatory documents in
% a safe, unless the documents are burnt completely.

%%%

(all d

(MEETING(d) & (Organizer(MEETING(d))) & After(MEETING(d)) &
Element_Of_MeetingDocuments(d,
Prepatory_Documents(MEETING(d)))

->

Obliged_To_Keep_In_Safe(Organizer(MEETING(d)), (Element_Of(d,
Contents(Safe(s))))

)

).

%%%

% Rule 10

% The organizer of any meeting is obliged to establish

% a list of all participants before that meeting.

%%%

(all d

(MEETING(d) & (Organizer(MEETING(d))) & Before(MEETING(d))

->

Obliged_To_Establish_List(Organizer(MEETING(d)),
Establish_Participants_List(MEETING(d)))

)

).

%%%

% Rule 11

% An agent is obliged to work a classified document at the secret level in a

% protected area.

%%%

(all d

(Agent(d) & Classified_Document(d) & (Classification(d) = Secret) &
During_Exec(Agent(d), Elaborate(d))

```

->
    ( During_Exec(Agent(d), Work(Protected_Area(d))))
)
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rule 12
% After the destruction of a classified document at the secret level,
% the document holder is obliged to inform the author of the document.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
(all d
    (Classified_Document(d) & (Classification(d) = Secret) &
      Holder(d) & Exec(Holder(d) , Destroy(d)) & (Transmitter(d))
    ->
      Obligated_To_Notify(Holder(d), Notify(Transmitter(d), Destroy(d)))
    )
).
%%%
%%% Implicit policies and real world facts
%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Real World Fact 1
% If there is a classified document, then this document
% has a classification level.
(all d
    (Classified_Document(d)
    ->
      ( exists level (Classification(d) = level))
    )
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

% Real World Fact 2

% All holders of a document is a organization agent.

% has a classification level.

(all d

(Holder(d)

->

Agent(d)

)

).

%%%

% Real World Fact 3

% All agents works for organization and a member

% of organization's employees.

(all d

(Agent(d)

->

Works_For(Agent(d),Employees(o))

)

).

%%%

% Real World Fact 4

% All transmitters are not organization agents.

(all d

(Transmitter(d)

->

(Works_For(Transmitter(d),Employees(o))) | (Transmitter(d) != Agent(d))

)

).

%%%

% Real World Fact 5

% Secret classified document is also a document.

(all d (

Classification(d) = Secret


```

->
Classified_Document(d)
)
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Real World Fact 6
% All representatives works for the organization.
% and they work in the sending office where the
% document originated.
(all d (
Representative(d)
->
Works_For(Representative(d),Sending_Office(o))
)
).
(all d (
Representative(d)
->
Works_For(Representative(d),Organization(o))
)
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Real World Fact 7
% Agents, Heads, Representatives, Holders,
% meeting Organizers, Actors, Witnesses are members of the organizations,
% i.e.they are valid employees.

(all d (
(Representative(d) | Agent(d) | Head(d) | Holder(d) |
Actor(d) | Witness(d) | Organizer(MEETING(d)))
->
Employees(o)

```

```

    )
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Real World Fact 8
% If there is a consignment note
% attached to the document than it is
% a classified document.
(all d
  (Consignment_Note(d)
    ->
    (exists d (Classified_Document(d)
      )))
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Real World Fact 9
% Destroying a classified document
% is to incinerate that document.
% After incineration, it becomes destroyed object.
(all d
  (DESTROY_DOCUMENT(d)
    ->
    Incinerate(d)
  )
).
(all d
  (After(Incinerate(d))
    ->
    DESTROY_DOCUMENT(d)
  )
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Real World Fact 10
% If the agent is not permitted to ask for revision of the classification

```

% level of the document, then
 % the agent is not forbidden to change the classification of the document.
 (all d
 (Classified_Document(d) & Transmitter(d) & Holder(d) &
 -Permitted_To_Ask_Revision(
 Holder(d),
 Ask(Transmitter(d), Change_Classification(d))
)
 ->
 Forbidden_To_Change_Classification(Holder(d), Change_Classification(d))
)
).

%%%

% Set of Support

%%%

(exists d Classified_Document(d)).
 (exists d Transmitter(d)).
 (exists d Evaluate_Classification(d)).
 (exists d Classification(d)).
 (exists d Representative(d)).
 (exists d Holder(d)).
 (exists o Organization(o)).
 (exists d Employees(d)).
 (exists d Document_Preservation(d)).
 (exists d Sending_Office(d)).
 (exists d Establish_Consignment_Note(d)).
 (exists d Consignment_Note(d)).
 (exists d Protected_Area(d)).
 (exists d Person(d)).
 (exists d Actor(d)).
 (exists d Destroyed_Object(d)).

(exists d Witness(d)).
(exists d Destruction_Minutes(d)).
(exists d MEETING(d)).
(exists d Organizer(d)).
(exists d Preparatory_Documents(d)).
(exists d Safe(d)).
(exists d Contents(d)).
(exists d Agent(d)).

%%%

% INSERT QUERY HERE

%%%

%%%

% END OF SET OF SUPPORT

%

%%%

end_of_list.

APPENDIX D QUERIES AND PROOFS

%%%

% QUERY I

%%%

-(exists d (

(Representative(d) | Holder(d) |

Witness(d) | Agent(d)|Organizer(MEETING(d)))

->

Employees(o)

)).

%%%

% PROOFS FOR Q_I

%%%

----- PROOF -----

19 [] -Agent(x16)|Works_For(Agent(x16),Employees(o)).

59 [] Agent(x26).

60 [] -Works_For(Agent(x26),Employees(o)).

69 [hyper,19,59] Works_For(Agent(x),Employees(o)).

70 [binary,69.1,60.1] \$F.

----- end of proof -----

----- PROOF -----

19 [] -Agent(x16)|Works_For(Agent(x16),Employees(o)).

59 [] Agent(x26).

60 [] -Works_For(Agent(x26),Employees(o)).

69 [hyper,19,59] Works_For(Agent(x),Employees(o)).

71 [hyper,69,60] \$F.

----- end of proof -----

----- statistics -----

clauses given	40
clauses generated	26
hyper_res generated	8
para_from generated	8
para_into generated	6
ur_res generated	4

```

demod & eval rewrites      0
clauses wt,lit,sk delete   0
tautologies deleted        1
clauses forward subsumed   15
(subsumed by sos)          12
Kbytes malloced            159
user CPU time               0.69      (0 hr, 0 min, 0 sec)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      QUERY II
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-(exists o ( exists d (
    Organization(o) & Sending_Office(o) & Classified_Document(d) &
    Decide_To_Send(o, Decide( Send(d))) & Consignment_Note(d)
    ->
    Obligated_To_Sign_Note(Head(Sending_Office(o)),
    Sign(Consignment_Note(d)))
    ))
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      PROOFS FOR QII
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
----- PROOF -----
6 [] -Organization(x6)| -Sending_Office(x6)| -Classified_Document($f3(x6))|
-Decide_To_Send(x6,Decide(Send($f3(x6))))|
-Consignment_Note($f3(x6))|
Obligated_To_Sign_Note(Head(Sending_Office(x6)),
Sign(Consignment_Note($f3(x6)))).
59 [] Organization(x26).
60 [] Sending_Office(x26).
61 [] Classified_Document(x27).
62 [] Decide_To_Send(x26,Decide(Send(x27))).
63 [] Consignment_Note(x27).
64[]-Obligated_To_Sign_Note(Head(Sending_Office(x26)),
Sign(Consignment_Note(x27))).

```

131[hyper,6,59,60,61,62,63] Obligated_To_Sign_Note(Head(Sending_Office(x)),
Sign(Consignment_Note(\$f3(x)))).

132 [binary,131.1,64.1] \$F.

----- end of proof -----

----- PROOF -----

6 [] -Organization(x6)| -Sending_Office(x6)| -Classified_Document(\$f3(x6))|
-Decide_To_Send(x6,Decide(Send(\$f3(x6))))|
-Consignment_Note(\$f3(x6))|Obligated_To_Sign_Note(
Head(Sending_Office(x6)),Sign(Consignment_Note(\$f3(x6)))).

59 [] Organization(x26).

60 [] Sending_Office(x26).

61 [] Classified_Document(x27).

62 [] Decide_To_Send(x26,Decide(Send(x27))).

63 [] Consignment_Note(x27).

64[]-Obligated_To_Sign_Note(Head(Sending_Office(x26)),
Sign(Consignment_Note(x27))).

131[hyper,6,59,60,61,62,63] Obligated_To_Sign_Note(Head(Sending_Office(x)),
Sign(Consignment_Note(\$f3(x)))).

133 [hyper,131,64] \$F.

----- end of proof -----

----- statistics -----

clauses given	81	
clauses generated	291	
hyper_res generated	19	
para_from generated	58	
para_into generated	207	
ur_res generated	7	
demod & eval rewrites	0	
clauses wt,lit,sk delete	0	
tautologies deleted	1	
clauses forward subsumed	222	
(subsumed by sos)	106	
Kbytes malloced	223	
user CPU time	0.92	(0 hr, 0 min, 0 sec)

%%%%%%%%%

% QUERY III

%%

-(exists d

(Agent(d) & Classified_Document(d) & (Classification(d) = Secret) &

During_Exec(Agent(d), Elaborate(d))

->

(During_Exec(Agent(d), Work(Protected_Area(d))))

)

).

%%

% PROOFS FOR Q_{III}

%%

----- PROOF -----

15 [] -Agent(x12)| -Classified_Document(x12)|Classification(x12)!=Secret| -
During_Exec(Agent(x12),Elaborate(x12))|During_Exec(Agent(x12),
Work(Protected_Area(x12))).

59 [] Agent(x26).

60 [] Classified_Document(x26).

61 [] Classification(x26)=Secret.

62 [] During_Exec(Agent(x26),Elaborate(x26)).

63 [] -During_Exec(Agent(x26),Work(Protected_Area(x26))).

88 [hyper,15,59,60,61,62] During_Exec(Agent(x),Work(Protected_Area(x))).

89 [binary,88.1,63.1] \$F.

----- end of proof -----

----- PROOF -----

15 [] -Agent(x12)| -Classified_Document(x12)|Classification(x12)!=Secret| -
During_Exec(Agent(x12),Elaborate(x12))|During_Exec(Agent(x12),
Work(Protected_Area(x12))).

59 [] Agent(x26).

60 [] Classified_Document(x26).

61 [] Classification(x26)=Secret.

62 [] During_Exec(Agent(x26),Elaborate(x26)).

63 [] -During_Exec(Agent(x26),Work(Protected_Area(x26))).

88 [hyper,15,59,60,61,62] During_Exec(Agent(x),Work(Protected_Area(x))).

90 [hyper,88,63] \$F.

----- end of proof -----

----- statistics -----

clauses given	68	
clauses generated	264	
hyper_res generated	16	
para_from generated	121	
para_into generated	122	
r_res generated	5	
demod & eval rewrites	0	
clauses wt,lit,sk delete	0	
tautologies deleted	0	
clauses forward subsumed	238	
(subsumed by sos)	31	
(subsumed by sos)	106	
Kbytes malloced	191	
user CPU time	0.81	(0 hr, 0 min, 0 sec)

%%

% QUERY IV

%%

-(exists d

 (Classified_Document(d)

 ->

 -(exists level (Evaluate_Classification(d) != level

)))

).

%%

% PROOFS FOR Q_{IV}

% OTTER DOES NOT REACH ANY PROOF FOR Q_{IV} .

%%

%%

% QUERY V

%%

-(exists d

 (Classified_Document(d) & (Classification(d) = Secret) & Holder(d) &

 DESTROY_DOCUMENT(d) & Actor(d) & Destroyed_Object(d) &

```

Witness(d) & After(Destroyed_Object(d)) & Destruction_Minutes(d)
->
Obliged_To_Sign_Destruction(Witness(d), Sign(Destruction_Minutes(d)))
)
).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      PROOFS FOR Qv
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
----- PROOF -----
11 [] -Classified_Document(x8)|Classification(x8)!=Secret| -Holder(x8)| -
DESTROY_DOCUMENT(x8)| -Actor(x8)| -Destroyed_Object(x8)| -Witness(x8)|
-After(Destroyed_Object(x8))|
Destruction_Minutes(x8)|Obliged_To_Sign_Destruction(Witness(x8),
Sign(Destruction_Minutes(x8))).
59 [] Classified_Document(x26).
60 [] Classification(x26)=Secret.
61 [] Holder(x26).
62 [] DESTROY_DOCUMENT(x26).
63 [] Actor(x26).
64 [] Destroyed_Object(x26).
65 [] Witness(x26).
66 [] After(Destroyed_Object(x26)).
67 [] Destruction_Minutes(x26).
68[]-Obliged_To_Sign_Destruction(Witness(x26),
Sign(Destruction_Minutes(x26))).
252[hyper,11,59,60,61,62,63,64,65,66,67]
Obliged_To_Sign_Destruction(Witness(x),Sign(Destruction_Minutes(x))).
253 [binary,252.1,68.1] $F.
----- end of proof -----
----- PROOF -----

11 [] -Classified_Document(x8)|Classification(x8)!=Secret| -Holder(x8)|
-DESTROY_DOCUMENT(x8)| -Actor(x8)| -Destroyed_Object(x8)|

```

-Witness(x8)|-After(Destroyed_Object(x8))|-Destruction_Minutes(x8)|
 Obligated_To_Sign_Destruction(Witness(x8),Sign(Destruction_Minutes(x8))).

59 [] Classified_Document(x26).

60 [] Classification(x26)=Secret.

61 [] Holder(x26).

62 [] DESTROY_DOCUMENT(x26).

63 [] Actor(x26).

64 [] Destroyed_Object(x26).

65 [] Witness(x26).

66 [] After(Destroyed_Object(x26)).

67 [] Destruction_Minutes(x26).

68[]-Obligated_To_Sign_Destruction(Witness(x26),
 Sign(Destruction_Minutes(x26))).

252[hyper,11,59,60,61,62,63,64,65,66,67]

Obligated_To_Sign_Destruction(Witness(x),
 Sign(Destruction_Minutes(x))).

254 [hyper,252,68] \$F.

----- end of proof -----

----- statistics -----

clauses given	121	
clauses generated	1071	
hyper_res generated	32	
para_from generated	299	
para_into generated	735	
ur_res generated	5	
demod & eval rewrites	0	
clauses wt,lit,sk delete	0	
tautologies deleted	0	
clauses forward subsumed	886	
(subsumed by sos)	332	
(subsumed by sos)	31	
(subsumed by sos)	106	
Kbytes malloced	351	
user CPU time	1.67	(0 hr, 0 min, 1 sec)

%%%%%%%%%

% QUERY VI

%%%%%%%%%

```

-(exists o ( exists d (
    Organization(o) & Sending_Office(o) & Classified_Document(d) &
    Decide_To_Send(o, Decide( Send(d))) & Consignment_Note(d) &
    Representative(d) &
    Works_For(Representative(d),Head(Sending_Office(o)))
    ->
    Permitted_To_Delegate_Sign(Head(Sending_Office(o)),
    Delegate(Representative(d),Sign(Consignment_Note(d))))
    ))
).

```

%%%

% PROOFS FOR Q_v

%%%

----- PROOF -----

```

7 [] -Organization(x7)| -Sending_Office(x7)| -Classified_Document($f4(x7))|
-Decide_To_Send(x7,Decide(Send($f4(x7))))| -Consignment_Note($f4(x7))|
-Representative($f4(x7))|
-Works_For(Representative($f4(x7)),Head(Sending_Office(x7)))|
Permitted_To_Delegate_Sign(Head(Sending_Office(x7)),
Delegate(Representative($f4(x7)),Sign(Consignment_Note($f4(x7))))).
59 [] Organization(x26).
60 [] Sending_Office(x26).
61 [] Classified_Document(x27).
62 [] Decide_To_Send(x26,Decide(Send(x27))).
63 [] Consignment_Note(x27).
64 [] Representative(x27).
65 [] Works_For(Representative(x27),Head(Sending_Office(x26))).
66 [] -Permitted_To_Delegate_Sign(Head(Sending_Office(x26)),
Delegate(Representative(x27),Sign(Consignment_Note(x27)))).
370[hyper,7,59,60,61,62,63,64,65]Permitted_To_Delegate_Sign(
Head(Sending_Office(x)),
Delegate(Representative($f4(x)),Sign(Consignment_Note($f4(x))))).

```

```

371 [binary,370.1,66.1] $F.
----- end of proof -----
----- PROOF -----
7 [] -Organization(x7)| -Sending_Office(x7)| -Classified_Document($f4(x7))|
-Decide_To_Send(x7,Decide(Send($f4(x7))))|
-Consignment_Note($f4(x7))|-Representative($f4(x7))|
-Works_For(Representative($f4(x7)),Head(Sending_Office(x7)))|
Permitted_To_Delegate_Sign(Head(Sending_Office(x7)),
Delegate(Representative($f4(x7)),Sign(Consignment_Note($f4(x7))))).
59 [] Organization(x26).
60 [] Sending_Office(x26).
61 [] Classified_Document(x27).
62 [] Decide_To_Send(x26,Decide(Send(x27))).
63 [] Consignment_Note(x27).
64 [] Representative(x27).
65 [] Works_For(Representative(x27),Head(Sending_Office(x26))).
66[]-Permitted_To_Delegate_Sign(Head(Sending_Office(x26)),
Delegate(Representative(x27),Sign(Consignment_Note(x27)))).
370[hyper,7,59,60,61,62,63,64,65]
Permitted_To_Delegate_Sign(Head(Sending_Office(x)),
Delegate(Representative($f4(x)),Sign(Consignment_Note($f4(x))))).
372 [hyper,370,66] $F.

```

```

----- end of proof -----
----- statistics -----

```

clauses given	125
clauses generated	1382
hyper_res generated	25
para_from generated	213
para_into generated	1139
ur_res generated	5
demod & eval rewrites	0
clauses wt,lit,sk delete	0
tautologies deleted	4
clauses forward subsumed	1073
(subsumed by sos)	592
(subsumed by sos)	332
(subsumed by sos)	31

(subsumed by sos)	106	
Kbytes malloced	574	
user CPU time	5.27	(0 hr, 0 min, 5 sec)

APPENDIX E IMPLEMENTATION DIAGRAMS

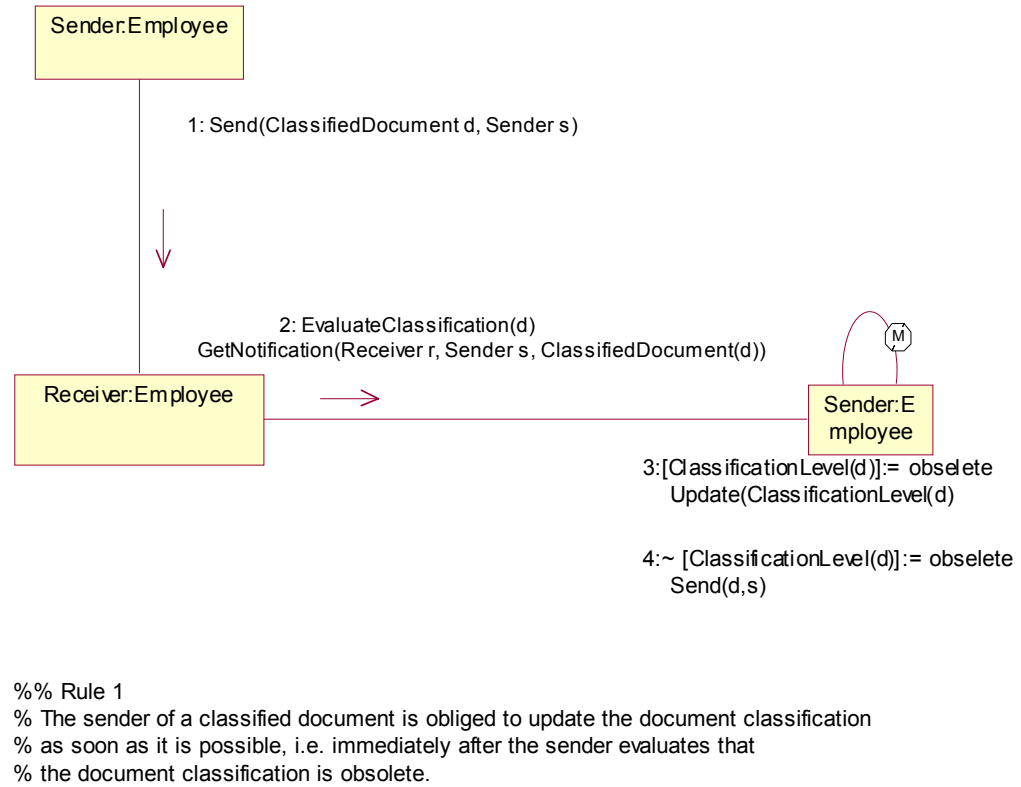
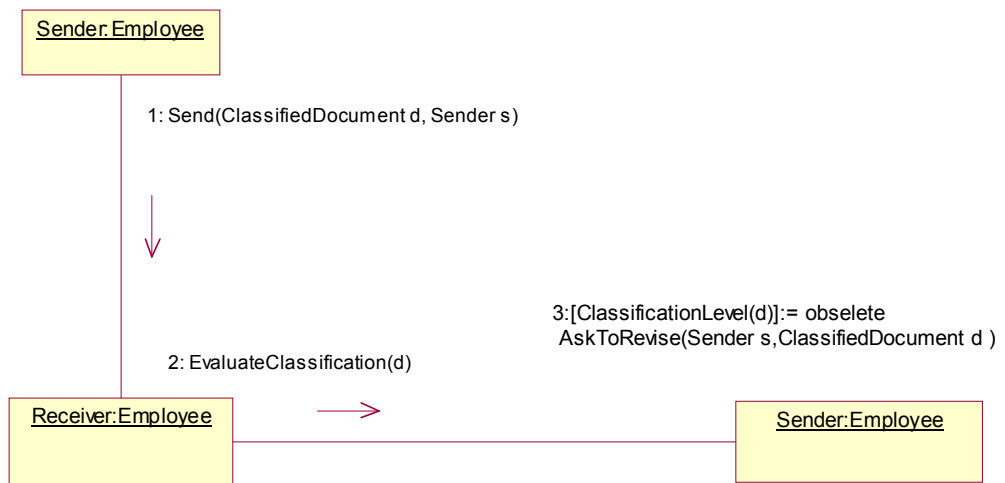


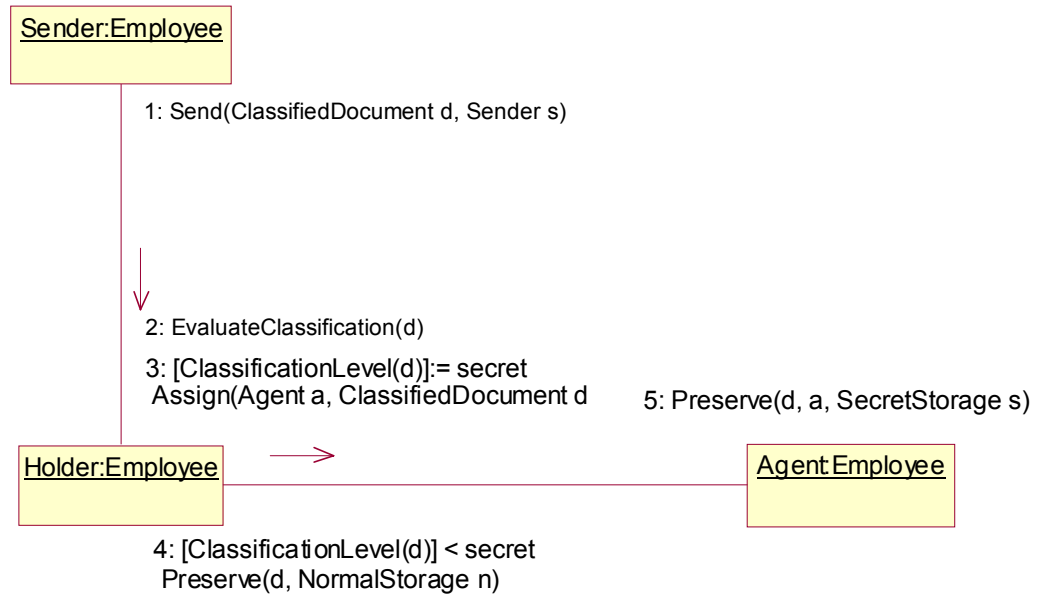
Figure 15. Rule 1 and Rule 2 Collaboration Diagram.



```

%%Rule 3
% The holder of a classified document is permitted to ask the sender to revise
% the classification of this document.
  
```

Figure 16. Rule 3 Collaboration Diagram.



```

%%Rule 4
% Every organization which holds some secret documents is
% obliged to
% designate an agent who is responsible for preserving
% these documents
  
```

Figure 17. Rule 4 Collaboration Diagram.

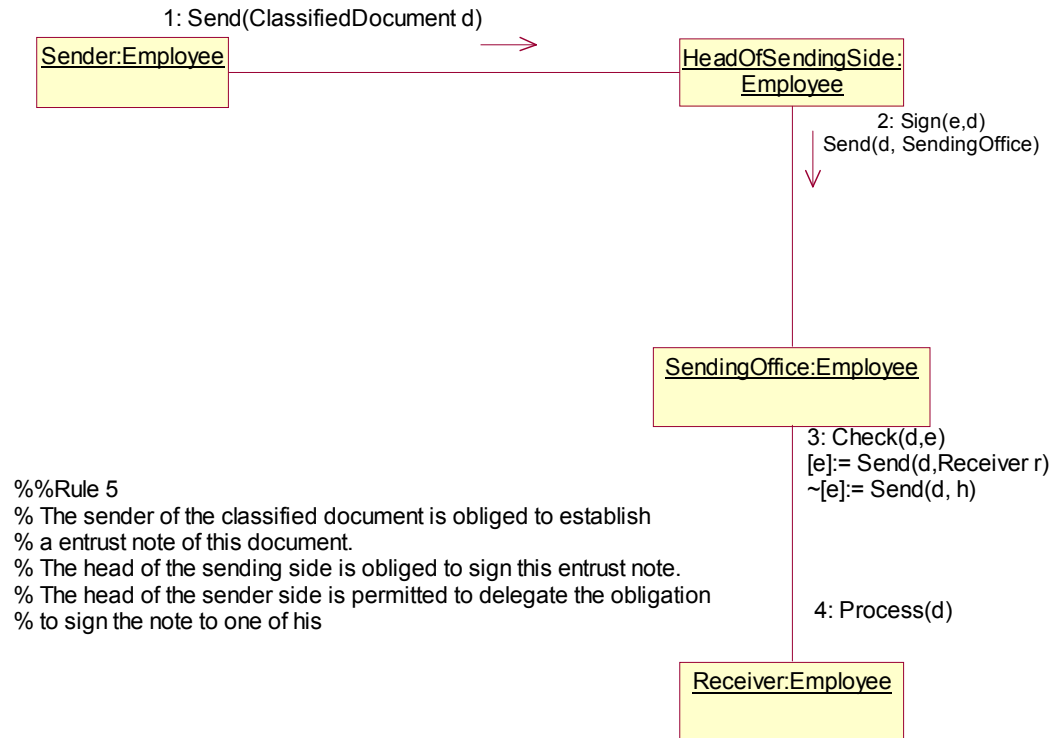
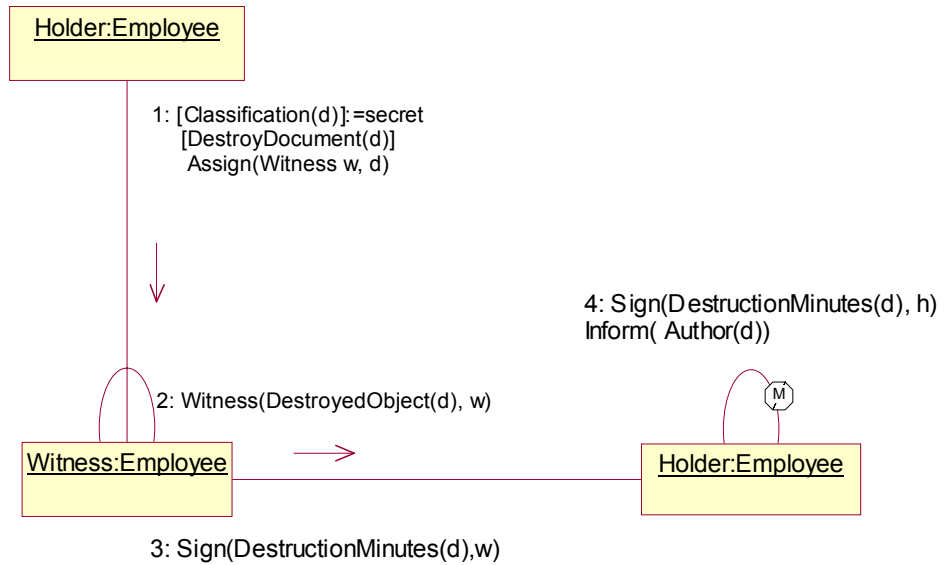


Figure 18. Rule 5 Collaboration Diagram.



%% Rule 7
 %The holder of the secret document and the witness of the
 destruction are obliged to
 %sign destruction time at every secret level classified document

Figure 19. Rule 7 Collaboration Diagram.

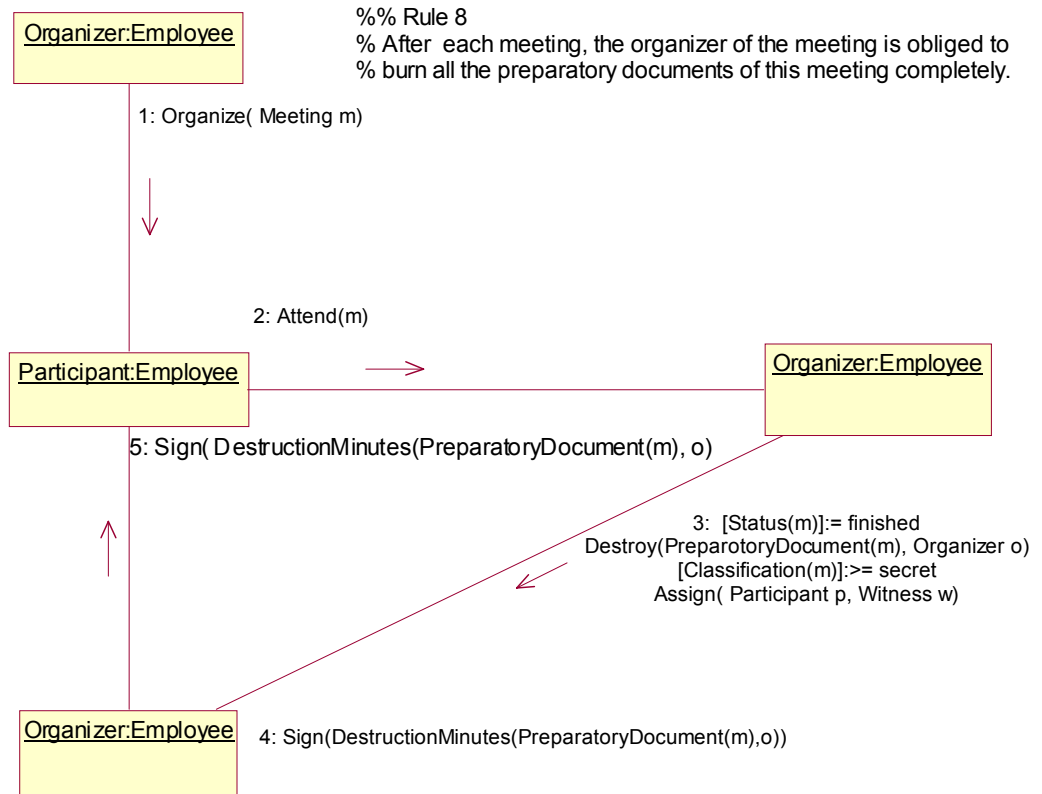


Figure 20. Rule 8 Collaboration Diagram.

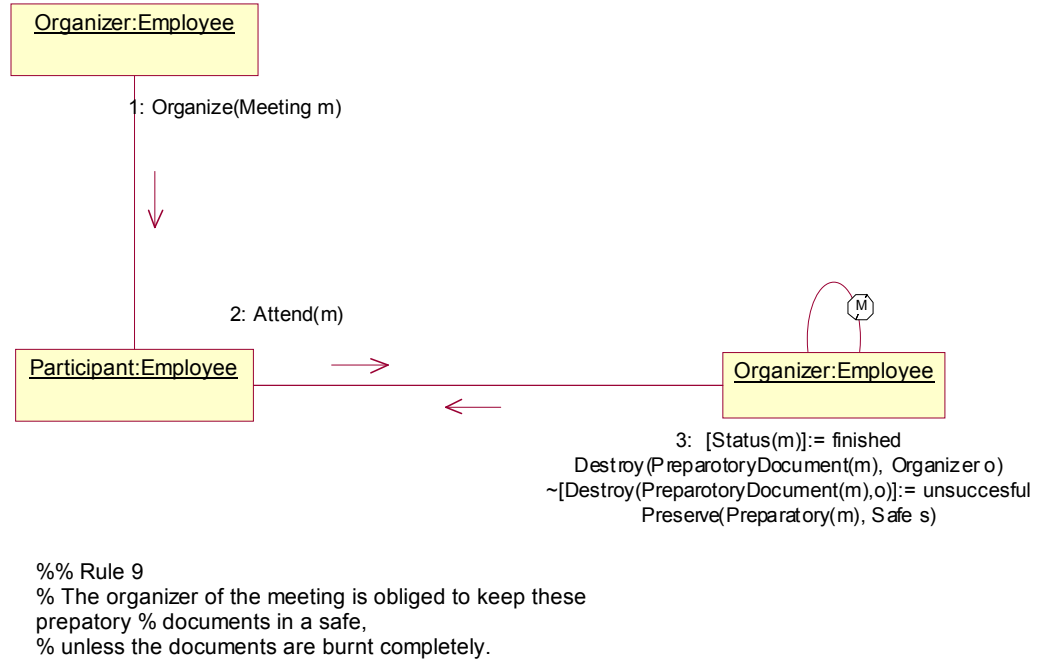


Figure 21. Rule 9 Collaboration Diagram.

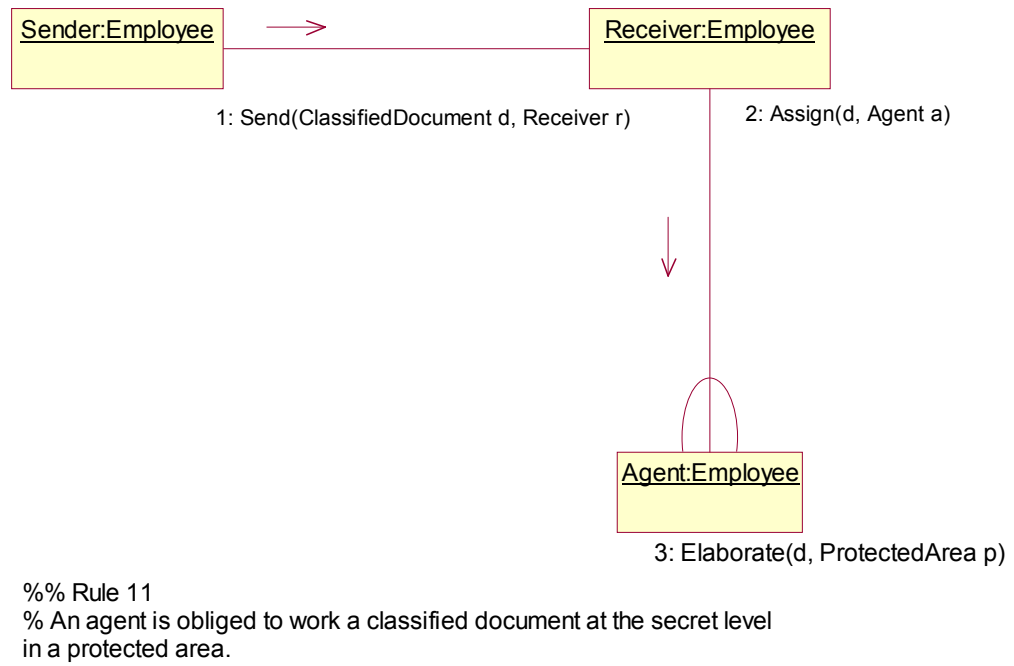


Figure 22. Rule 11 Collaboration Diagram.

APPENDIX F GLOSSARY

Automated Testing: The automated generation and execution of test plans and test cases from both models of policy and queries about policy.

Collaboration: A description of a collection of objects that interact to implement some behavior within the context. Tersely, collaboration directly shows the implementation of an operation. Collaboration diagram is a graphical representation of collaboration.

Community: A set of interacting objects that have come together into a configuration so that they can interact to achieve some purpose

Configuration File: A script containing a test plan that specifies what is being tested, test cases and the steps for executing the test plan.

Deontic logic: A universal formal language, useful for expressing any kind of knowledge or data.

Meta policy: Policy about policy.

Normative System: A system in which the behavior of the interacting objects or agents governed by a set of norms.

Obligation: A prescription that particular behavior is required. An obligation is fulfilled by the occurrence of the prescribed behavior.

OTTER: A first-order logic with equality theorem prover.

Permission: A prescription that a particular behavior is allowed to occur. Also permission is equivalent to there being no obligation for the behavior not to occur.

Policy: Statements of goals, or rules or guidance governing the actions taken to satisfy goals[See the Michael-Ong-Rowe for more on policy,36].

Policy Base: A database of policies.

Policy Workbench: A set of tools for formalizing and assessing policy. In other words, an integrated set of computer-based tools for developing, reasoning about, and maintaining and enforcing policy.

Prohibition: A prescription that particular behavior must not occur. Also prohibition is equivalent to there being an obligation for the behavior not to occur

Schema-based Approach: The creation of policy from a model.

Set of support strategy: A strategy that keys on the denial clause or clauses. It seeks a contradiction that gives the program a convenient termination.

Test criteria: A rule or collection of rules that impose test requirements on a set of test cases.

Test case: A set of inputs, execution conditions, and expected results developed for a particular objective.

UML: Unified Modeling Language is a language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems.

LIST OF REFERENCES

1. Michael J.B., Sibley E.H., Baum R.F., Li F. On the Axiomization of Security Policy: Some Tentative Observations About Logic Representation. In *Proceedings of the 6th IFIP Working Conference on Database Security*, IEEE (1992), 401-429.
2. Cuppens F., Saurel C. Specifying a Security Policy: A Case Study. In *Proceeding of CSFW (Computer Security Foundations Workshop)*, 9th IEEE (1996), 123 -134.
3. Moffet. J., Soloman M. Policy hierarchies for distributed systems management. *IEEE Journal on Selected Areas of Communications(JSAC)*, Special Issue on Network Management, 11(9), December 1993.1404-1414.
4. Cuppens F., Cholvy L., Saurel C., Carrere J. Merging Regulations: Analysis of a Practical Example. In *Proceedings of Fifth International Conference on Artificial Intelligence and Law*, (College Park, Md. 1995), 123-136.
5. Schmid M., Ghosh A., Hill F. Techniques for Evaluating the Robustness of Windows NT Software. In *Proceedings of DARPA Information Survivability Conference and Exposition*. Vol.2 IEEE (1999), 347-360 .
6. Kim Y., Carlson R.C. Scenario Based Integration Testing for Object-Oriented Software Development. In *Proceedings of Eighth Asian Test Symposium*, IEEE (Shanghai, Nov. 1999), 283 -288.
7. Putman J. Model for Fault Tolerance and Policy from RM-ODP Expressed in UML/OCL. *Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time Distributed Computing*, IEEE (Newport Beach, Calif., March 2000), 189 -196.
8. Linington F.P. Options for Expressing ODP Enterprise Communities and Their Policies by Using UML. In *Proceedings of Enterprise Distributed Object Computing Conference*, IEEE(University of Mannheim, Germany September1999), 72 -82.
9. Voas M.J., Miller W.K. The Revealing Power of a Test Case. *Journal of Software Testing, Verification, and Reliability*, (May 1992) 2(1), 25-42.
10. Evans S.A. Reasoning with UML Class Diagrams. In *Proceedings of the Second Workshop on Industrial Strength Formal Specification Techniques*, IEEE(Florida, October 1998), 102-113.
11. Favre, L., Clerici, S. Integrating UML and algebraic specification techniques. In *Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages*, IEEE Computer Society(TOOLS-32 Pacific'99), 151-162.
12. Selonen P., Koskimes K.,Sakkinen M. How to Make Apples from Oranges in UML. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, (Maui, Hawai,3–6 January 2001), 86-95.
13. Shroff M., France B.R. Towards a Formalization of UML Class Structures in Z. In *Proceedings of the Computer Software and Applications Conference*, IEEE(Washington D.C., August 1997), 646 –651.
14. Selic B. A Generic Framework for Modeling Resources with UML. *IEEE Computer* Vol.33 No.6 (2000), 64-69.

15. Linington P., Milosevic Z., and Raymond K. Policies in Communities: Extending the ODP Enterprise Viewpoint. In *Proceedings of the Enterprise Distributed Object Computing Workshop*, IEEE(San Diego, November 1998), 14 –24.
16. Kim S.K., Carrington D. Visualization of Formal Specifications. In *Proceedings of the Sixth Asia Pacific Software Engineering Conference*, IEEE(Takamatsu, Japan, December 1999), 102 -109.
17. Rumbaugh J., Jacobson I. *The Unified Modeling Language Reference Manual*. Addison Wesley, Reading, 1999.
18. Ober I. More Meaningful UML Models. . In *Proceedings of the 37th International Technology of Object-Oriented Languages and Systems Conference*, IEEE(Sydney, Australia, November 2000), 146 -157.
19. Linington P., Milosevic Z., and Raymond K. Policies in Communities: Extending the ODP Enterprise Viewpoint. In *Proceedings of the Enterprise Distributed Object Computing Workshop*, IEEE Computer Society Press(San Diego, November 1998), 14 –24.
20. ISO/IEC IS 10746-2 International Standard 10746-2 ITU-T Recommendation X.902: Open Distributed Processing – Reference Model- Part 2: Foundations, 1995.
21. Michael B.J. *A Formal Process for Testing the Consistency of Composed Security Policies*. Ph.D. dissertation, George Mason University, 1993,6.
22. Pffleger, C. P. *Security in Computing*. Prentice Hall, 1999.
23. Riehle R. Software Risk Assessment Lecture Notes. Naval Postgraduate School, Monterey, Calif., 2000.
24. IEEE Standard for Software Test Documentation, IEEE Std 829-1998, Institute of Electrical and Electronics Engineers Inc., 1998, 2.
25. Binder, R.V *Testing Object Oriented Systems*. Addison Wesley, 1999.
26. Myers, G.J. *The Art of Software Testing*. Wiley-Interscience Publication, 1979,10.
27. Kaner C., Falk J, Nguyen Q.H. *Testing Computer Software*. Wiley Computer Publishing, 1999, 249.
28. McCune W. *Otter 3.0 Reference Manual and Guide*. Argonne National Laboratory, ftp://info.mcs.anl.gov/pub/Otter/Papers/otter3_manual.pdf
29. Wos L., Overbeek L., and Boyle J. *Automated Reasoning: Introduction and Applications*. McGraw-Hill, 1992.
30. Kalman A.J. *Automated Reasoning with OTTER*. Rinton Press, 2001.
31. Abdurazik A., Offutt J. Using UML Collaboration Diagrams for Static Checking and Test Generation. *Lecture Notes in Computer Science* Vol. 1939(2000), Springer Press, 383-395.
32. Engels G., Groenewegen L.P., and Kappal G. Object-oriented Specification of Coordinated Collaboration. In *Proceedings of the IFIP World Conference on IT Tools*, IEEE(Canberra, Australia, September 1996), 437-449.
33. Kaner C., Lawrence B., *The Los Altos Workshop on Software Testing (LAWST) Handbook* .2000. www.kaner.com
34. Pettichord B., Three Keys to Test Automation, *Journal of Stickyminds.com*, December 2000.

35. Arnold T., Building an Automation Framework with Rational Visual Test. December 1997.
<http://www.data-dimensions.com/Testers'Network/autorationvt1.htm>
36. Michael J.B., Ong V.L., Rowe N.C., Natural-language Processing Support for Developing Policy-governed Software Systems, in *Proceedings of the Thirty Ninth International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, IEEE Computer Society Press (Santa Barbara, Calif., July 2001), 263-274.
37. Lie K., Ong T., A Modeling Approach for Handling Business Rules and Exceptions. *The Computer Journal*, Vol.42, No.3 (1999).
38. Stamper R.K., Signs, Information, Norms, and Systems. In *Proceedings of the Sign of Work: Semiotics and Information Processing In Organizations*, (Berlin, de Gruyter, 1996) 349-399.
39. Vanderveen K.B., Ramamoorthy C.V. Anytime Reasoning in First-Order Logic. In *Proceedings of the Ninth Tools with Artificial Intelligence Conference*, IEEE (Newport Beach, Calif., November 1997), 142 -148.
40. Khresiat L., Dalal Mukesh. Anytime Reasoning With Probabilistic Inequalities. In *Proceedings of the Ninth Tools with Artificial Intelligence*, IEEE (Newport Beach, Calif., November 1997), 60–66.
41. Sturgill D.B., Segre A.M. Using Hundreds of Workstations to Solve First-order Logic Problems. In *Twelfth National Conference on Artificial Intelligence*, (Cambridge, Massachusetts, July 1994), 187-192.
42. Russel S., Norvig P. Artificial Intelligence: A Modern Approach. Prentice-Hall, 1995.
43. Horvitz E.J. Reasoning Under Varying and Uncertain Resource Constraints. In *Proceedings Eleventh International Joint Conference on Artificial Intelligence (IJCAI)*, (Detroit, August 1989), 1121-1127.
44. Ramoni M., Riva A. Belief Maintenance in Bayesian Networks. In *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-1994)*, pp.204-212.
45. Haddawy P., Frisch A.M. Anytime Deduction for Probabilistic Logic. *Artificial Intelligence*, 69(1-2): 93, 1994.
46. Hosmer H.H. Security is fuzzy. In *Proceedings of New Security Paradigms Workshop (SIGSAC)*, ACM Press (Fairfax, Virginia, November 1993), 175 – 184.
47. Zadeh A.L. Fuzzy Logic, Neural Networks, and Soft Computing. *Communications of the ACM*, March 1994/Vol.37.No.3.77-84.
48. Zadeh L.A. The Concept of a Linguistic Variable. Fuzzy Sets and Applications: Selected Papers by L.A.Zadeh, ed by Yager, Ovchinnikov, R.M. Tong, and H.T.Nguyen, John Wiley and Sons, 1987.
49. Zimmerman H.J. Fuzzy Sets, Decision-Making and Expert Systems, Kluwer Academic Publishers, 1987, 73.
50. McAllister M.L., Dockery J., Ovchinnikov S., Adlassnig K.P. Tutorial on Fuzzy Logic in Simulation. In *Proceedings of the Winter Simulation Conference*, ACM (San Diego, Calif., December 1985), 40-44.

51. Rosch E. Cognitive Representations of Semantic Categories. *Journal of Experimental Psychology: General*, 1975(104), 532-547.
52. Sibley, E.H., Michael, J.B., Wexelblat, R.L. Use of Experimental Policy Workbench: Description and Preliminary Results. In Landwehr, C.E. and Jajodia, S., eds *Database Security, V: Status and Prospect*. Elsevier Science(North-Holland), Amsterdam, 1992, 47-76.
53. Sloane, S.B. The Use of Artificial Intelligence by the United States Navy: Case Study of a Failure. *AI Magazine* 12, 1(Spring 1991), 80-92.
54. Arsanji A. Rule Pattern Language 2001: A Pattern Language for Adaptive Manners and Scalable Business Rule Design and Construction. In *Proceedings of the Technology of Object-oriented Languages and Systems(TOOLS)*, IEEE Computer Society(Monterey, Calif., July 2001), 369-376.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Prof. James Bret Michael
Naval Postgraduate School

4. Prof. Richard Riehle
Naval Postgraduate School

5. Chairman
Naval Postgraduate School

6. Kara Kuvvetleri Komutanligi(Turkish Army Headquarters)
Bakanliklar-Ankara/Turkey

7. Kara Harp Okulu Komutanligi(Turkish Army Military Academy)
Bakanliklar-Ankara/Turkey

8. Lt. Mehmet Sezgin
Bakanliklar-Ankara/Turkey

9. US Military Academy
West Point, NY

10. US Naval Academy
Annapolis, MD

11. Middle East Technical University
Balgat-Ankara/Turkey

12. Bilkent University
Balgat-Ankara/Turkey

13. Marmara University
Goztepe Kampusu
Goztepe-Istanbul/Turkey

14. Bogazici University
Bebek Istanbul/Turkey

15. Harp Akademileri Komutanligi
4.Levent Istanbul/Turkey

16. Kara Harp Okulu Dekanligi
Bakanliklar/Ankara/Turkey

17. Gazi Universitesi
Besevler/Ankara/Turkey

18. Hacettepe Universitesi
Beytepe/Ankara/Turkey

19. Bilkent Library Information Services System(BLISS)
Balgat/Ankara/Turkey

20. Yeditepe Universitesi
Istanbul/Turkey

21. Prof. Edgar Sibley
George Mason University

22. Dr. Larry Wos
Argonne National Labaratory

23. Dr. Greg Larsen
Institute for Defense Analysis

24. Mr. Terry Mayfield
Institute for Defense Analysis

25. Prof. Neil Rowe
Naval Postgraduate School

26. Dr. Bruce Barnes
George Mason University

27. Prof. Michael Beeson
San Jose State University

28. Dr. William McCune
Argonne National Laboratory

29. Prof. V.S. Alagar
Concordia University

30. Ms. Barbara Jones Redmon
George Mason University

31. Prof. Morris Sloman
Imperial College
